Szakdolgozat

Krakomperger Róbert

Debrecen
2007

Debreceni Egyetem

Informatikai Kar

Java alapú webtechnológiák a gyakorlatban

Témavezető:

Dr. Juhász István
egyetemi adjunktus

Készítette:

Krakomperger Róbert
programtervező informatikus (BSc.)

Debrecen
2007

Tartalomjegyzék

Bevezetés

Egyetemi tanulmányaim alatt, miután megismertem a Java alapjait, szinte folyamatosan foglalkoztam – elsősorban Java alapú – webes technológiákkal. Nemcsak a hallgatott hasonló témákat felölelő tárgyak keretében, hanem az egyetemi szférán kívül is: olykor egyszerűen az érdeklődés vezérelt, máskor munkavállalás céljából. Ez alatt az idő alatt kialakult némi rálátásom a ma használatos eszközökre, a webalkalmazásokkal szemben támasztott elvárásokra, követelményekre. Megismertem néhány technológiát, keretrendszert és egyéb eszközöket. Ezek közé tartozik például a JMS, JSP, JSF, Beehive, XML, Hibernate, különböző integrált fejlesztői környezetek (Eclipse, NetBeans) stb. Ezen ismereteim nagy részét egy-egy alkalmazás fejlesztésén keresztül próbáltam mélyíteni. Mindez motivált abban, hogy szakdolgozatomat hasonló témakörben készítsem el.

Dolgozatomban egy Java alapú webalkalmazás létrehozásához szükséges néhány technológiát, eszközrendszert szeretnék bemutatni, melyek vizsgálatát egy alkalmazás elkészítésén keresztül végeztem. A technológiák nagy részét már korábbi munkáim során is használtam, de van néhány kivétel, mint például az AJAX támogatást biztosító Ajax4jsf eszközrendszer. Ennek oka az, hogy próbáltam olyan technológiákat kiválasztani, melyek esetében rendelkezem némi tapasztalattal egyrészt azért, hogy ne apró problémák megoldása – melyek elenyésző jelentőségük ellenére napokon keresztül képesek lefoglalni a gyakorló fejlesztő kapacitását – vegye el a fejlesztési idő nagy részét – érthető okok miatt –, másrészt pedig azért, hogy autentikus elemzéssel szolgálhassak egy-egy technológia vonatkozásában. Ennek ellenére mégis választottam néhány számomra eddig nem ismert eszközt, elsősorban ismereteim bővítése céljából, másrészt pedig azért, mert esetlegesen a mai webes elvárások indokolják; ide ismét az AJAX-ot hoznám példaként.

Az első néhány fejezetben egy-egy az általam készített webalkalmazásban használt – egyébként ma is fejlődő, jelenleg a webszférában joggal helyet foglaló – technológiát fogok bemutatni. Ezek vonatkozásában törekszem majd arra, hogy áttekintést nyújtsak, részletekbe menő elemzést csak nagyon ritkán és indokolt esetben prezentálok. Természetesen az irodalmjegyzék tartalmazza a megfelelő hivatkozásokat, melyeken keresztül a részletes információk is elérhetők. Ily módon dolgozatom akár kiindulópontot is jelenthet egy-egy olyan érdeklődő számára, aki a feldolgozott témával ismerkedni szeretne, esetleg a Java alapú webes technológiák mélyebb megismerését célozza. Továbbá a függelék és egyes helyeken a dokumentáció is több architektúrális és egyéb modellt, ábrát tartalmaz, melyek a könnyebb megértést segítik.

A technológiák jellemzése után kitérek a fejlesztés menetére, és néhány olyan általam

fontosnak vélt problémát mutatok be, melyek munkám során felmerültek, fejtörést okoztak, esetleg adott eszközrendszer hiányosságának vélem. Ezek kiküszöbölésére megpróbálok egy-egy javaslatot tenni, valamint kitérek az általam alkalmazott megoldásra.

Célom egyrészt az, hogy a technológiákról megfelelő prezentációt nyújtsak, valamint szeretnék egy értékelést bemutatni az alkalmazott eszközök/technológiák használhatóságáról, integrálhatóságáról, egymással való kompatibilitásáról. Másrészt, mintegy járulékos haszonként saját ismereteim mélyítését és bővítését is szeretném e dolgozat elkészítésével elérni.

I. JavaServer Faces technológia

A JavaServer Faces technológia Java alapú webalkalmazások fejlesztését támogató, JSP alapú, MVC tervezési minta mentén implementált keretrendszer.

Két fő komponensből áll:

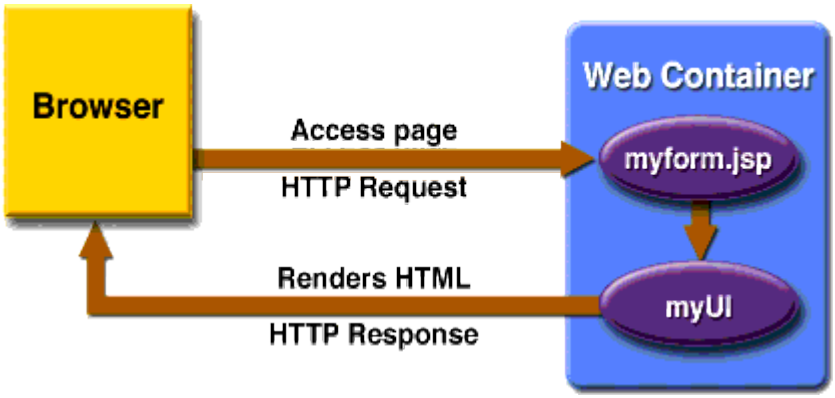
- Felhasználói interfész-komponensek (továbbiakban UI komponens) megjelenítésére és kezelésére szolgáló API, mely eszközöket nyújt események menedzselésére, szerveroldali validációra és adatkonverzióra, az oldalak közötti navigációra, a hozzáférhetőség és többnyelvű alkalmazás készítésének támogatására, valamint mindezen eszközök kibővítésére.
- Két JSP tag library az UI komponensek leírására, és azok szerveroldali objektumokkal történő összerendelésére.

A jólformált programozási modell és tag library meglehetősen lecsökkenti egy webalkalmazás elkészítésével és karbantartásával járó nehézségeket.

A JSF használatával egyszerűen köthetünk egy a kliens által kiváltott eseményt egy szerveroldali kódhoz (például egy JavaBean egy metódusához). Lehetőségünk van

- UI komponenst tárolni egy-egy szerveroldali objektum attribútumaként - így is segítve a dinamikus oldalak létrehozását.
- saját, újrafelhasználható és kiterjeszthető UI komponensek létrehozására
- az UI komponensek állapotának tárolására nemcsak egy requesten (kérésen) belül.

A JSF működése egy egyszerű kérés esetén:



A myform.jsp egy JSF oldal, mely JSF tageket tartalmazó JSP oldal. A myUI kezeli a JSP oldal által hivatkozott objektumokat: az UI komponens-, az ezeken regisztrált eseményfigyelő- (event listerner), validátor- és konverter-, valamint a komponensek alkalmazásspecifikus adatait és funkcionalitásait reprezentáló objektumokat.

A JSF talán legfontosabb erénye, hogy jól szeparálható a logikai és megjelenítési réteg. Az egész keretrendszer e vonalon lett felépítve, és mintegy ki is kényszeríti a fejlesztőből ugyanezt egy-egy alkalmazás elkészítése során. Ennek következtében további előny, hogy egy fejlesztői csapat tagjai könnyedén koncentrálhatnak saját feladatukra, hiszen azok jól elkülönülnek egymástól.

Egy JSF alkalmazás, mint általában minden Java alapú webalkalmazás egy servlet containerben fut, és a következőket tartalmazza:

- alkalmazásspecifikus adatokat és funkcionalitást reprezentáló JavaBeanek
- event listeners (esemény figyelők)
- weboldalak (pl. JSP)
- szerveroldali osztályok, például egy adatbázis eléréséhez

Amit a JSF mindezeken felül nyújt:

- JSP tag library az UI komponensek megjelenítéséhez
- JSP tag library az események kezeléséhez, validáláshoz és egyéb funkciók megvalósításához
- UI komponenseket állapotaikkal együtt reprezentáló szerveroldali objektumok
- ún. Backing beanek, melyek az UI komponensek attribútumait és funkcióit definiálják
- validátorok, konverterek, eseményfigyelők és -kezelők
- egy konfigurációs fájl, mely az alkalmazással kapcsolatos beállításokat tartalmazza (ez általában a *faces-config.xml*)

Egy olyan JSF alkalmazásnak, mely JSP oldalak segítségével generál HTML-t, tartalmaznia kell két tag libraryt. Egyet az UI komponenseket reprezentáló tagek definiálására, valamint egy másikat a validátorok, eseménykezelők és egyéb funkciók megvalósítására. A JavaServer Faces implementáció mindkét tag libraryt biztosítja.

Egy JSF alkalmazás elkészítésének főbb lépései:

- Az oldalak létrehozása az UI komponensek felhasználásával.
- Az oldalak közötti navigáció leírása az alkalmazás konfigurációs állományában.
- A “backing bean”-ek fejlesztése.
- A kezelt beanek deklarálása az alkalmazás konfigurációs állományában.

A fenti lépések végezhetők párhuzamosan és tetszőleges sorrendben néhány kivételtől eltekintve. Például ha egy fejlesztői csapat dolgozik az alkalmazáson, a weboldalak fejlesztőjének/fejlesztőinek ismernie kell a backing beanek nevét ahhoz, hogy hivatkozhassa azokat az oldalakról.

Felhasználói interfész-komponens (UI komponens) modell:

Az UI komponensek konfigurálható és újrafelhasználható elemek, melyek egy JSF alkalmazás felhasználói interfészét alkotják. Egy komponens lehet egyszerű, mint egy nyomógomb (HTML button), vagy összetett, mint egy táblázat (HTML table), ami több komponensből épül fel.

A JSF technológia UI komponensek gazdag, konfigurálható választékát nyújta a fejlesztők számára:

- UI komponens osztályok/interfészek: az UI komponensek viselkedését és állapotait definiálják (attribútumok tárolása, események kezelése). Ezek az osztályok és interfészek természetesen kiterjeszthetők, újraimplementálhatók, íly módon saját UI komponensek könnyedén definiálhatók.
  Minden UI komponens kiterjeszti az *UIComponentBase* osztályt, ami az alapértelmezett állapot és viselkedésmódot hivatott definiálni. Továbbá az UI komponens osztályok implementálnak egy vagy több viselkedési interfészt.
- Komponens megjelenítési modell (rendering model): a komponensek renderelésére különálló *renderer* definiálható. Ebből kifolyólag több renderer is megadható egy komponenshez, így az többféleképpen jeleníthető meg akár egyazon kliens böngészőjén is.
  A JSP oldalak és az alkalmazás fejlesztői megváltoztathatják egy-egy komponens kinézetét azáltal, hogy azt a taget használják, mely a komponens és a renderer megfelelő kombinációját reprezentálja.
  Az úgynevezett *render kit* határozza meg a komponens osztályok leképezését komponens tagekre egy-egy kliensnek megfelelően. A JSF implementáció tartalmaz egy standard HTML render kitet a HTML kliensek részére.
  A render kit minden egyes támogatott UI komponens számára definiál többféle renderer osztályt, melyek különféle módon jelenítenek meg egy-egy komponenst. Például az `UISelectOne` komponensnek 3 renderer osztálya van, melyek rádiógombként, combo boxként, vagy list boxként jelenítik meg azt. Ezek funkcionalitása egyezik, viszont máshogy jelennek meg a felhasználó böngészőjén. *(Függelék I.1)*
- Konvertálási modell: Egy JSF alkalmazásban lehetőségünk van egy-egy komponenst hozzárendelni valamely szerveroldali objektumhoz, például egy backing beanhez. Egy ilyen összerendelés esetén az alkalmazás szempontjából a komponensek adatainak kétféle nézetét különböztethetjük meg:
  - Model nézetben az adatokra azok konkrét típusát figyelembe véve tekintünk (egész, hosszú egész stb.)

- Megjelenítési szinten a felhasználó által olvasható és módosítható formában tekintünk az adatokra. Például egy dátum típusú adat reprezentálható 3 sztringként év-hónap-nap formában.

A JSF implementáció automatikus konverziót végez a fenti két nézet között, ha az adott komponenssel összerendelt bean-attribútum típusa megfelel a komponens aktuális értékével. Például ha egy `UISelectBoolean` komponens mögött egy `java.lang.Boolean` típusú attribútum áll, a komponens által tartalmazott `java.lang.String` automatikusan `java.lang.Boolean` típusra konvertálódik.

A JSF lehetőséget biztosít saját konverterek létrehozására is, melyek egy-egy `UIOutput` komponensen regisztrálhatók, így a model és megjelenítési nézet között tetszőleges konverzió végezhető.

- *Esemény* és *Figyelő* modell: Egy `Event` objektum ismeri az őt kiváltó komponenst valamint információkat tárol magáról az eseményről. Az alkalmazásnak szolgáltatnia kell egy `Listener` implementációt, és regisztrálni azt egy komponensen. A felhasználó egy komponens aktiválásával (például egy nyomógombra való kattintás) vált ki egy eseményt, melynek hatására a JSF meghívja az eseményt feldolgozó metódust.

  A JSF három féle eseményt különböztet meg: value-change events (érétkváltozás), action events (akció), data-model events (adatmodell esemény).

  Egy action event akkor váltódik ki, ha a felhasználó egy olyan komponenst aktivál, amely implementálja az `ActionSource` *viselkedési interfészt*. Ilyen például egy hyperlink, vagy egy nyomógomb.

  A value-change event azt jelzi, ha egy `UIInput` vagy annak egy alosztályát reprezentáló komponens aktuális értéke megváltozik. Például egy *checkbox* átállítása.

  Data-model event kiváltódását `UIData` komponens egy új sorának kiválasztása okozza. A data-model események tárgyalása túlmutat ezen leírás keretein.

  Egy standard komponens által kiváltott esemény alkalmazásspecifikus kezelésére két lehetőség áll rendelkezésre:

  - Egy eseményfigyelő osztály implementálása és a komponens beágyazása egy `valueChangeListener` vagy egy `actionListener` tag közé, ezzel regisztrálva az event listenert.
  - Egy az eseményt kezelő metódus implementálása valamely backing beanben, amire a figyelendő komponens egy attribútumával hivatkozhatunk (*method binding*).

- Validációs modell: A JSF lehetővé teszi a szerkeszthető input komponensek (például szöveg

mező) validálását. E mechanizmus még azelőtt zajlik, hogy a megjelenítési és a model nézet közötti konverzió megtörténne.

Ahogyan a konverziós modellben, ebben az esetben is rendelkezésre áll néhány beépített osztály az alapvető validációk elvégzésére, valamint a JSF standard tag library tartalmaz néhány taget a standard validátorok implementációinak hivatkozására.

Továbbá a legtöbb tag rendelkezik néhány attribútummal, melyek a validátorok konfigurációját hivatottak beállítani. Ilyen például egy *text field* elfogadható minimum vagy maximum értéke szám esetén, vagy a maximális hossz sztring esetén.

A validációs modell megengedi saját validátor létrehozását, és lehetővé teszi annak használatát megfelelő tag biztosításával. Erre két lehetőség áll rendelkezésre:

- A `Validator` interfész implementálása és regisztrálása az alkalmazásban, valamint saját tag létrehozása vagy a `validator` tag használata, hogy a vizsgálandó taget a megfelelő validátor ellenőrizze.
- Egy validátor metódus implementálásával valamely backing beanben, melyre az ellenőrzött tag `validator` attribútumával hivatkozhatunk.

- Navigációs modell: *(Függelék I.2)* A JSF szempontjából az egyes oldalak közötti navigáció lényegében szabályok egy halmaza, ami meghatározza, hogy egy hyperlink vagy egy nyomógomb lenyomása után mely oldal kerül megjelenítésre. A szabályokat az alkalmazás konfigurációs állományában kell rögzíteni; erre XML elemek állnak rendelkezésre.

  Legegyszerűbben a szabályok megadásával és ezek – a hyperlinkek és nyomógombok `action` attribútumán keresztül történő – hivatkozásával valósíthatunk meg navigációt egy alkalmazás keretein belül. Az `action` attribútum lényegében egy kimenő sztring, amely alapján a JSF implementáció megkeresi a megfelelő szabályt az alkalmazás konfigurációs állományában, és az abban leírtaknak megfelelően cselekszik. Adott oldalról származó kimenő sztring alapján adott oldalt jelenít meg.

  Komplikáltabb alkalmazások esetén szükség lehet navigációt megvalósító metódusokra a backing beanekben. A metódus visszatérési típusa és paraméterei a JSF által rögzítettek. Ekkor az `action` attribútum egy *method-binding*et (metódus hivatkozást) tartalmaz. Mikor a komponens aktiválásra kerül, egy action-event váltódik ki, melyet az alapértelmezett `ActionListener` kezel oly módon, hogy meghívja az `action` attribútum által hivatkozott metódust. Ennek visszatérési értéke szolgáltatja a kimenő sztringet, mely alapján a JSF implementáció meghatározza a következő megjelenítendő oldalt.

Backing beanek kezelése:

Webalkalmazások készítésének egy kritikus pontja az erőforrások megfelelő kezelése. Ez magában foglalja az UI komponenseket és alkalmazásspecifikus folyamatokat reprezentáló objektumok megfelelő szeparálását, valamint ezen objektumok élettartamának (*request scope, session scope, application scope*) megfelelő kezelését.

Egy tipikus JSF alkalmazás tartalmaz egy vagy több, az UI komponensekkel összerendelt backing beant (ezek JavaBeans elemek), melyek az UI komponensek egyes attribútumainak, értékének tártolását hivatottak szolgálni, esetleg nyújtanak navigációt, validációt vagy eseménykezelést megvalósító funkciókat.

Backing bean attribútumok és metódusok JSF oldalról történő referálására a JSF expression language (EL – kifejezés nyelv) nyújt lehetőséget. Egy EL kifejezés a `#{` és `}` többkarakteres szimbólum között állhat, sztringként. Tartalmazhat metódus- és attribútumhivatkozást, literált és operátort. További információk az EL kifejezésekről http://java.sun.com/j2ee/1.4/docs/tutorial/doc/ oldalon találhatók.

Többnyelvű alkalmazások:

A JSF technológián keresztül könnyedén készíthetünk többnyelvű alkalmazásokat, melyet `ResourceBundle` (Standard J2SE eszköz) objektumok használatval valósíthatunk meg. Az alkalmazott `ResourceBundle` objektumokat regisztrálnunk kell az alkalmazás konfigurációs állományában, továbbá itt adhatjuk meg a támogatott nyelveket és az alapértelmezett *locale* beállítást. A JSF alapértelemzésben a `PropertyResourceBundle` használatát támogatja. Ekkor különböző, kulcs-érték párokat tartalmazó szöveges állományokat kell létrehozni, alkalmazkodva a megadott névkonveciókhoz (`<ResourceBundleName>_<Locale>.properties`).

Ahhoz, hogy egy JSF oldalról a *bundle* elérhető legyen, regisztrálnunk kell azt az adott oldalon is. Ezek után EL kifejezés segítségével hivatkozhatunk a megadott kulcsokon keresztül a *locale*-nak megfelelő szövegre.

A kliens böngészőjének beállításai lekérdezésével megállapíthatjuk annak *locale* beállításait.

Egy JSF oldal életciklusa:

Egy JSF alkalmazás kétféle *requestet*(kérés) és kétféle *response*-t (válasz) ismer:

- *Faces* válasz: Egy *servlet response* amely a *Render Response Phase* (válasz renderelésének fázisa) során keletkezik.
- *Non-Faces* válasz: Egy *servlet response* amely nem a *Render Response Phase* során keletkezik. Ilyen lehet például egy nem JSF rendszerben létrehozott JSP által generált válasz.
- *Faces kérés*: Egy *servlet request*, amely egy korábban generált *Faces response*-tól származik.
- *Non-Faces* kérés: Egy *servlet request*, amely az alkalmazás egy olyan eleméhez érkezik, amely a JSF keretrendszer hatáskörén kívül esik (például egy servlet, vagy egy JSP oldal).

A standard kérés-feldolgozás életciklusa – *Faces request* generál *Faces response*-t *(Függelék I.3)*:

- *Restore View Phase* (nézet-visszaállítás): Akkor indul el, mikor egy link vagy egy nyomógomb kerül aktiválásra. Ebben a fázisban készül el az aktuális oldal nézete. Összerendelésre kerülnek az oldalon található komponensek, és a hozzájuk tartozó validátorok, eseménykezelők, valamint mentésre kerül a nézet a `FacesContext` példányban. A `FacesContext` tartalmazza egy egyszerű kérés feldolgozásához szükséges összes információt (paraméterek, attribútumok stb.).
- *Apply Request Values Phase* (kérésspecifikus értékek elfogadása): Minden megjelenítendő komponens újragenerálja a saját értékét a request-paraméterek alapján. Ha konverziós hiba keletkezik, egy hibaüzenet jön létre az adott komponenshez kapcsolódóan, mely a *Render Response Phase* során kerül megjelenítésre. A fázis végén a komponensek új értékeikkel rendelkeznek, valamint az újonnan keletkezett események és üzenetek pufferelésre kerülnek.
- *Process Validations Phase* (validációs fázis): E fázis során a JSF implementáció lefuttatja a megjelenítendő komponenseken regisztrált validátorokat. Az invalid értékekkel rendelkező komponensekkel az előző fázisban leírt módon jár el (hibaüzenet generálása és megjelenítése a *Render Response Phase* során).
- *Update Model Values Phase* (model-értékek érvényesítése): A valid értékeket a komponensekhez rendelt megfelelő szerveroldali objektumok attribútumai is átveszik.
- *Invoke Application Phase* (alkamazásszintű hívások): Ebben a fázisban az alkalmazásszintű események kerülnek kezelésre. Ilyen lehet például egy form elküldése.

- *Render Response Phase* (válaszgenerálás): Ezalatt a fázis alatt a vezérlés (vagyis az oldal újragenerálása) átkerül a JSP konténerre. Az oldalon elhelyezkedő komponensek megjelenítésre kerülnek, ahogy a JSP konténer bejárja a tageket. Amennyiben az *Apply Request Values Phase, Process Validations Phase,* vagy az *Update Model Values Phase* során valamilyen hiba történt, az eredeti oldal, valamint ha van pufferelt hibaüzenet, az is megjelenítésre kerül.

A JSF *Sun Microsystems* által létrehozott implementációján kívül több kiegészítés, és további JSF implementációk találhatók a weben, többek között az *Apache* vagy a *Jboss* gondozásában. Ezek használata további lehetőségeket biztosít a technológia alkalmazói számára.

II.1 Hibernate

A Hibernate egy Java és .NET alkalmazásokba ágyazható relációs/objektumrelációs perzisztenciakezelést megvalósító eszközrendszer. A Hibernate lehetővé teszi perzisztens objektumok létrehozását az objektumorientált paradigma mentén – beleértve az öröklődést, polimorfizmust, kompozíciót, asszociációt, kollekciókat. A Hibernate saját SQL kiegészítéssel rendelkezik (HQL – *Hibernate Query Language*), de támogatott a natív SQL nyelv használata is, valamint rendelkezésre bocsát egy `Criteria` kritériumrendszert is, mely segítségével objektumorientált módon fogalmazhatunk meg lekérdezéseket.

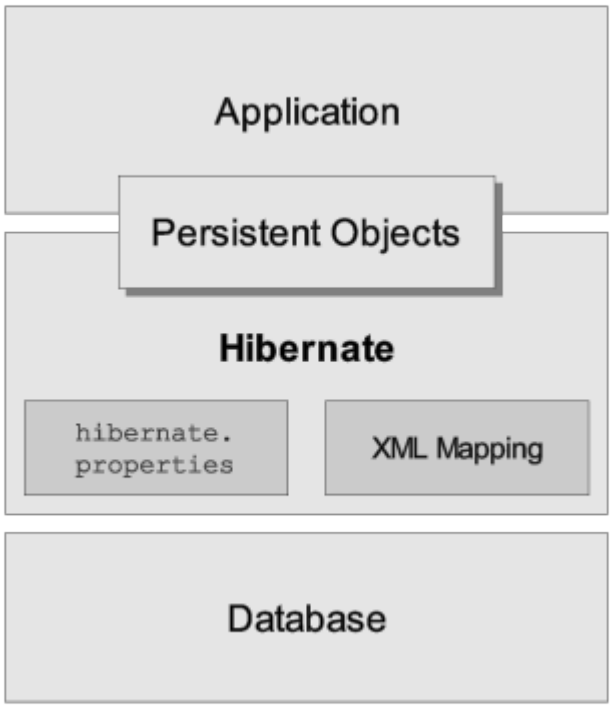
Kiemelendő, hogy a Hibernate nem rejti el az SQL nyújtotta lehetőségeket, tehát a relációs modell mentén szerzett tapasztalatok, az SQL nyelv minden ereje a Hibernate használata mellett is kiaknázható.

A Hibernate néhány alrendszere:

- *Hibernate Core*: SQL-szkriptek generálása, JDBC eredményhalmaz kezelése, objektum konverzió. Hordozhatóságot biztosít minden SQL adatbázis között. A perzisztens objektumok és az adatbázistáblák közötti leképezés XML állományokban definiált metaadatokon keresztül konfigurálható.
  A *Hibernate EntityManager*rel és *Hibernate Annotations*szel kombinálva hatékony Java perzisztenciakezelést valósíthatunk meg.
- *Hibernate Annotations:* Objektum-reláció leképezés JDK 5.0 annotációk segítségével. Kiegészítő vagy alternatív megoldás a *Hibernate Core* XML bázisú metaadatai mellett. Standard *Java Persistence* és *EJB 3.0* valamint Hibernate-specifikus annotációkat biztosít.
- *Hibernate EntityManager:* A *Hibernate EntityManager* implementálja a standard *Java Persistence* perzisztenciakezelő API-t, a statndard *Java Persistence Query Language*-et, a standard *Java Persistence* objektum-életciklusra vonatkozó szabályokat, a standard *Java Persistence* konfigurációt és csomagolást. A *Hibernate EntityManager* mellett bármikor használható a Hibernate natív API, natív SQL és natív JDBC.
- *NHibernate:* A *Hibernate Core* .NET adaptációja. Perzisztens .NET objektumok kezelését teszi lehetővé; az ehhez szükséges metaadatok XML-ben adhatók meg.
  - OO paradigma támogatása – öröklődés, polimorfizmus, kompozíció, .NET kollekciók
  - natív .NET
  - *open source (LGPL* – Lesser GNU Public License)

II.2 Hibernate Core és Hibernate Annotations

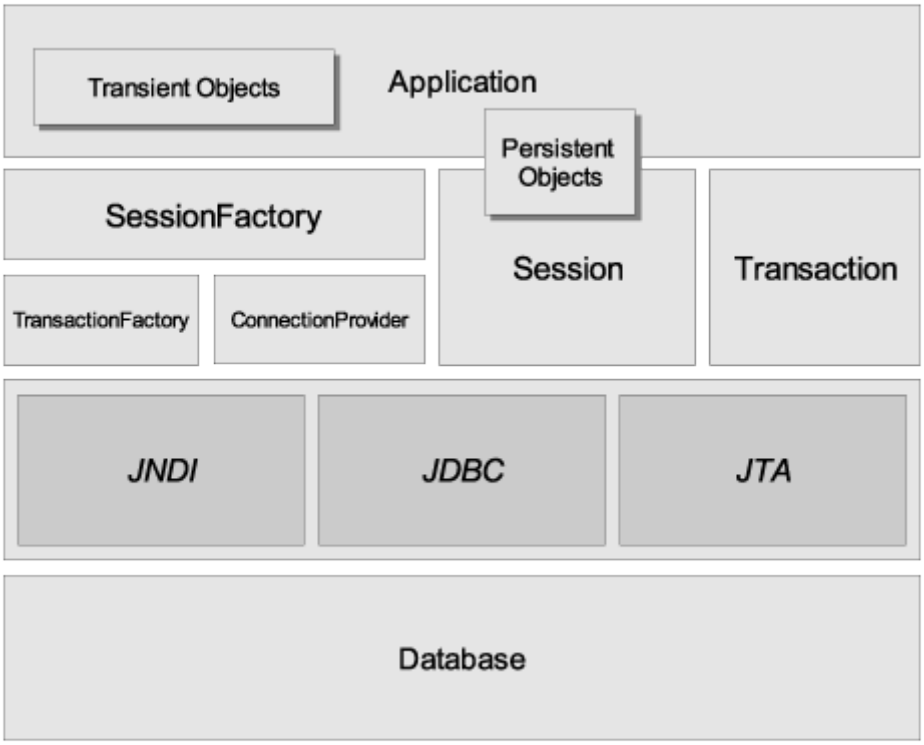
Koncepcionális Hibernate architektúra:



A fenti diagram a Hibernate Core felépítését hivatott bemutatni. A metaadatok és konfigurációs beállítások segítségével, az adatbázisban tárolt adatok alapján a Hibernate perzisztens objektumokat szolgáltat az alkalmazás számára, vagy az alkalmazás által létrehozott objektumokat helyezi el az adatbázisban.

A Hibernate Annotations beágyazásával a `hibernate.properties` konfigurációs állományt javallott `hibernate.cfg.xml` -re cserélni, melynek használatával az XML nyújtotta lehetőségeket is kihasználhatjuk, ezzel hatékonyabb és összetettebb beállításokat kialakítva. Továbbá az *XML mapping* mellett annotációkat is használhatunk az objektum-reláció leképezésekhez szükséges metaadatok megadásához.

Egy Hibernate-alkalmazás felépítése:



- `SessionFactory`: Egy állandó *cache* a már kialakított adatbázis-leképezés tárolására, kliens a `ConnectionProvider` felé valamint `Session`-öket hoz létre. Továbbá opcionálisan fenntarthat egy cache-t a tranzakciók között újrafelhasználható adatok számára.
- `Session`: Egy JDBC csatlakozást lefedő objektum, mely egy párbeszédet reprezentál az alkalmazás és a perzisztens tároló között, továbbá `Transaction` objektumokat kezel és Cache-t biztosít a perzisztens objektumok számára.
- *Persistent Objects*: Perzisztens állapotot és üzleti funkciókat megvalósító objektumok. Lehetnek *JavaBeans* vagy *POJO* (egyszerű Java objektum) objektumok, viszont speciálisak abból a szempontból, hogy pontosan egy *`Session`*-höz lehetnek hozzárendelve. Amint a `Session` lezárásra kerül, az alkalmazás szemszögéből újra hagyományos objektumokká válnak.
- *Transient objects*: Olyan objektumok, melyek adott pillanatban nincsnek `Session`-höz rendelve. Lehetnek az alkalmazás által példányosított nem perzisztens, vagy leendő perzisztens objektumok, esetleg egy már lezárt `Session` által létrehozott objektumok.
- `Transaction`: Szokásos tranzakció fogalom, atomosan végrehajtandó feladatsorozat. Lefedi a JDBC, JTA vagy CORBA tranzakciót. Egy `Session` több tranzakciót is magába foglalhat.

- `ConnectionProvider`: JDBC kapcsolatokat épít ki és kezel. Olyan általános interfészt biztosít az alkalmazás számára, mely elfedi `Datasource`-ot vagy `DriverManagert`; ezek az adatbázissal való, már implementációfüggő kommunikációt hivatottak biztosítani. Kiterjeszthető vagy újraimplementálható a fejlesztő által.
- `TransactionFactory`: `Transaction` példányokat állít elő. Kiterjeszthető vagy újraimplementálható a fejlesztő által.

A Hibernate több kiegészítő interfészt biztosít a programozó számára, melyek implementálásával egyedi, specializált perzisztenciakezelést valósíthat meg.

Konfiguráció:

A Hibernate egyik tervezési szempontja volt, hogy több különböző környezetben is működtethető legyen. Ennél fogva nagyon sok - a rendszer konfigurálását szolgáló - paraméterrel rendelkezik.

A Hibernate beállításait egy `org.hibernate.cfg.Configuration` példány reprezentálja, melynek attribútumait többféle módon inicializálhatjuk:

- Programkódból: A `Configuration` példány létrehozása után különböző beállító metódusokon keresztül manipulálhatjuk az egyes adattagokat, megadhatunk *XML mapping* állományokat, vagy perzisztens osztályokat, melyeket a Hibernate az alkalmazás működése közben kezelni fog.
- `hibernate.properties`: Egyfajta konfigurációs állomány, melyben kulcs-érték párokkal adhatjuk meg az egyes paraméterek értékét. Ha a `Configuration` példányosításakor nem adunk meg a konfigurációs állományra vonatkozó információkat, alapértelmezett módon a `hibernate.properties` fájlt fogja keresni a rendszer. Ebben az esetben az alkalmazásra vonatkozó CLASSPATH-ban láthatóvá kell tenni az állományt.
- `hibernate.cfg.xml`: <property> elemek segítségével adhatjuk meg az egyes paraméterek értékét.
- A `java` parancs használatával rendszerparaméterként
- `java.util.Properties` objektum átadása a `Configuration` példánynak

Amennyiben `Hibernate Annotations`-t is használunk, az `org.hibernate.cfg.AnnotationConfiguration` váltja fel a fenti `Configuration`-t *(Függelék II.2.1)*.

HQL (Hibernate Query Language): A Hibernate rendelkezik egy saját, szintaxis tekintetében az SQL-hez nagyon hasonló, teljes mértékben objektumorientált lekérdező nyelvvel.
Néhány fontosabb jellemzője:

- A kis- és nagybetűk nem különböznek, kivéve a Java osztályok és attribútumok neveit.
- A `FROM` záradék teljes mértékben hasonló az SQL-hez. Megadhatunk átnevezést, "`,`"-vel vagy kulcsszóval leírt `JOIN`-t, estleg alkérdést is. Viszont nem az adatbázisbeli táblaneveket, hanem a perzisztens Java osztályok teljesen minősített nevét kell megadnunk.
- A `SELECT` záradék úgyszintén hasonló az SQL-hez, viszont amennyiben nem akarunk szelekciót alkalmazni, tehát az összes attribútumot lekérdezzük, akkor a záradék elhagyható. Megadható attribútum, alkérdés és néhány aggregált függvény.
- A `WHERE` záradékban használható a legtöbb ismert SQL operátor (pl. `LIKE, IN, BETWEEN` stb.), néhány beépített függvény (`current_time`), JDBC-ből ismert `?` pozicionális paraméter, névvel rendelkező paraméter (`:name`), SQL literál, Java `public static final` konstans, alkérdés stb.
- Megadható `ORDER BY` és `GROUP BY` záradék.
- Polimorf lekérdezések: A `FROM` záradékban megadott osztály és az azt kiterjesztő osztályok minden példánya, valamint interfész minden implementáló osztályának példánya lekérdezésre kerül. Például egy `FROM java.lang.Object` lekérdezés minden perzisztens objektumot visszaad.
- A HQL támogat DML utasításokat is, mint `UPDATE`, `DELETE`, `INSERT...SELECT...`

Kritérium-lekérdezések:

Az `org.hibernate.Criteria` interfész egy lekérdezést reprezentál. `Criteria` példányok egy `Session` objektum által készíthetőek. A lekérdezés eredménye megszorítások hozáadásával szűkíthető, melyeket az `org.hibernate.criterion.Restrictions` állít elő statikus metódusok segítségével. Például a `Restrictions.like("name", "Ste%")` az SQL `LIKE` operátorát reprezentálja.

Példa egy Hibernate alkalmazás konfigurációjára:

1. Egy `hibernate.properties` vagy `hibernate.cfg.xml` *(Függelék II.2.2)* konfigurációs állomány elhelyezése az alkalmazásra vonatkozó CLASSPATH gyökerében, melyekben az alábbi beállítások megadása szükséges:
   - `driver_class`: az adatbáziskezelő-rendszer meghajtója, mely általában beszerezhető a DBMS weboldalán.
   - `dialect`: DBMS-specifikus dialektus. Ezeket a Hibernate definiálja, a legtöbb adatbáziskezelő-rendszerre vonatkozóan. A generálandó SQL utasítások szintaktikáját határozza meg.
   - `connection_datasource`: Akkor szükséges, ha JNDI név alapján szeretnénk az adatbázis-kapcsolatot elérni.
2. A perzisztens osztályok definiálása.
3. A definiált perzisztens osztályok leképezése relációkra (XML vagy annotációk segítségével – használhatók együttesen is, de egy osztálydefinícióban csak az egyik megoldás alkalmazható). Továbbá a létrehozott osztályokat regisztrálnunk kell a `Configuration` példány számára a fenti *Konfiguráció* pontban leírtaknak megfelelő módon.
4. Egy `Configuration` példány létrehozása. Ha `hibernate.cfg.xml`-t használunk konfigurációs állományként, akkor a `Configuration` példány `configure()` metódusának meghívása is szükséges.

Természetesen konfigurációs és egyéb lehetőségek tárháza áll rendelkezésünkre egy Hibernate alkalmazás létrehozása esetén. Ezek részletezése – sőt felsorolása is – túlmutat jelen dokumentáció keretein. Minden további információ megtalálható a www.hibernate.org weboldalon.

III.1 AJAX – Aszinkron JavaScript technológia és XML

Az AJAX egyfajta alkalmazásfejlesztési technika, mely néhány mára már érett technológiát ötvöz:

- megjelenítés HTML/XHTML és CSS használatával
- dinamikus elemek és interakció a *Document Object Model* (DOM) alapján
- adattovábbítás és -manipuláció XML és XSLT segítségével
- aszinkron adattovábbítás XMLHttpRequest-en keresztül
- és JavaScript mindezek összekapcsolására

Széles körben elfogadott az AJAX technológiaként való értelmezése is. Az egyszerűség kedvéért a következőkben eként fogok hivatkozni rá.

Fontos megjegyezni, hogy annak ellenére, hogy az AJAX mint elnevezés csak 2005-ben lett igazán elterjedt, nem egy új technológiáról van szó. Windows platformon már több éve elérhetők az AJAX nyújtotta lehetőségek az Internet Explorer alá fejlesztők számára. Valamint korábban a webfejlesztők *plug-in*ek, Java appletek és rejtett *frame*-ek különböző kombinációival emulálták a manapság az AJAX által nyújtott interakciós modellt.

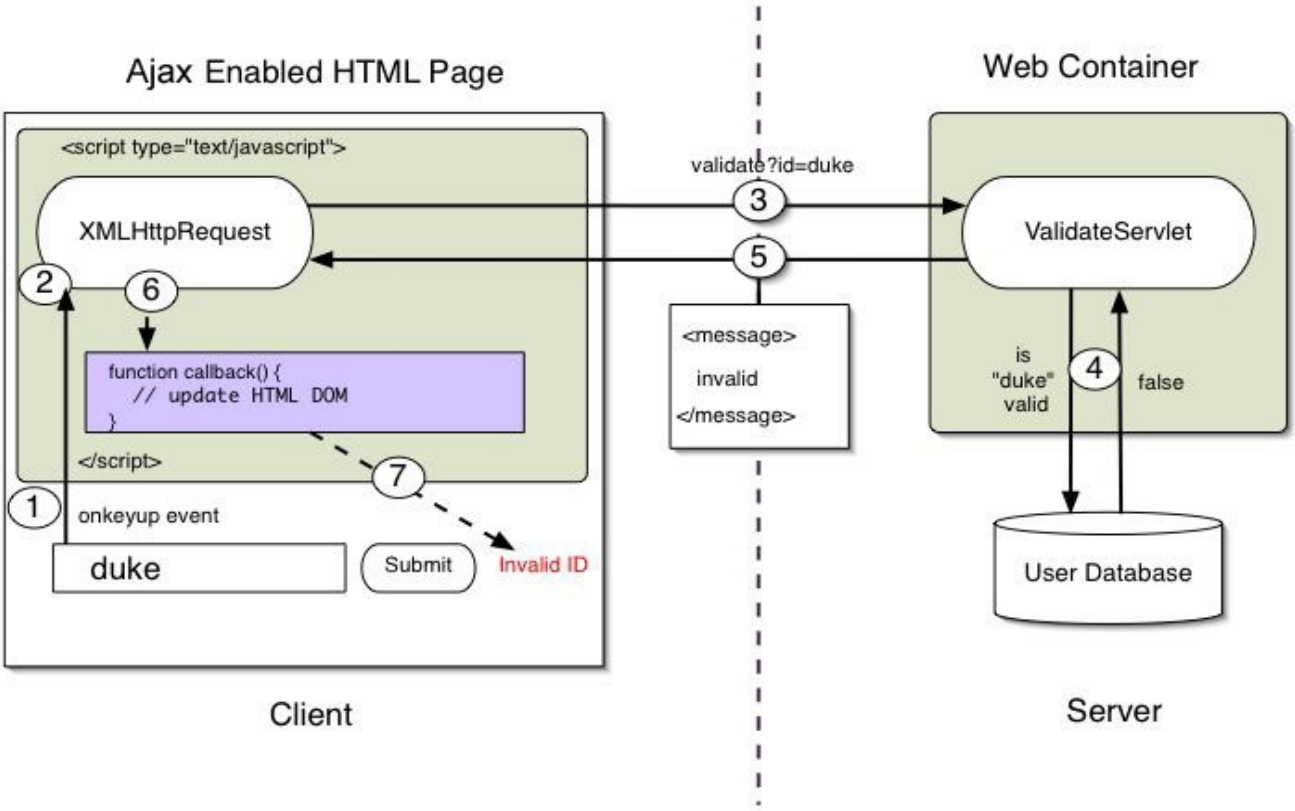
AJAX a hagyományokkal szemben *(Függelék III.1.1.a, Függelék III.1.1.b)*:

Egy klasszikus webalkalmazás-modellben a legtöbb felhasználói művelet egy *HTTP Reqestet* (HTTP Kérést) generál, melyet a webszerver dolgoz fel, majd válaszként egy HTML oldalt küld vissza a kliensnek. Ez a megközelítés nem nyújt igazi felhasználói élményt, mivel az egyes tevékenységek között a felhasználónak várakoznia kell. Egy AJAX alkalmazás megszünteti ezt a problémát azzal, hogy egy újabb réteget – az AJAX motort – ágyaz a kliens és a szerver közé. Ez első ránézésre még tovább csökkenti a felhasználó és az alkalmazás közötti intarakció dinamizmusát, viszont mégis ennek az ellenkezője történik.
A *session* elindulásakor az egész weboldal betöltése helyett a böngésző elindítja a JavaScriptben megírt AJAX motort, amely a felhasználói interfész elemeinek megjelenítését végzi és kommunikál a szerverrel a kliens által végrehajtott műveleteknek megfelelően. Aszinkron interakciót tesz lehetővé az alkalmazás és a felhasználó között a szerverrel való kommunikációtól függetlenül.

Minden felhasználói művelet *HTTP Request* generálása helyett egy JavaScript hívást indukál az AJAX motor felé. Minden olyan *Response*-t, mely nem igényel kommunikációt a szerver és a kliens között – például egyszerű adatvalidáció, memóriabeli adatkezelés valamint egyes navigációs esetek –, az AJAX motor hoz létre és kezel. Amennyiben a *HTTP Response* létrehozásához a

szerverre is szükség van – például ha a kérés adatok továbbítása szerveroldali feldolgozásra vagy további elemek kialakítása a felhasználói interfészen, estleg új adatok kérése a szervertől –, a motor a *Request*-et aszinkron kéréssé alakítja, leggyakrabban XML használatával.

AJAX egy Java alkalmazásban:



Az alkalmazás tartalmaz egy statikus vagy egy JSP által generált HTML oldalt. Ezen található egy olyan *form*, melynek validációja szerveroldali műveleten – melyet a `ValidateServlet` valósít meg – keresztül történik. Az adatellenőrzést az oldal újragenerálása nélkül szeretnénk végrehajtani.Az ábrán található lépések:

1. A felhasználó kivált egy eseményt valamilyen tevékenységével (fent: *onkeyup* – egy billentyű leütése)
2. Létrejön és beállítódik egy *XmlHttpRequest* objektum az eseménynek megfelelően
3. Az *XmlHttpRequest* végrehajt egy hívást a szerver felé a konfigurációja alapján
4. A kérést feldolgozza a `ValidateServlet`
5. A `ValidateServlet` visszaküldi az eredményt egy XML dokumentumban.
6. *Az XmlHttpRequest* objektum meghívja a `callback()` (a HTML DOM frissítése) függvényt és feldolgozza az eredményt.

III.2 Ajax4jsf

Az Ajax4jsf keretrendszer a JavaServer Faces egy kiterjesztése, melynek segítségével JavaScript kódok írása nélkül adhatunk AJAX funkciókat akár már elkészült JSF alapú alkalmazásainkhoz is. Támogatja a JSF életciklust, validációt, adatkonverziókat, a statikus és dinamikus erőforrások kezelésével együtt.

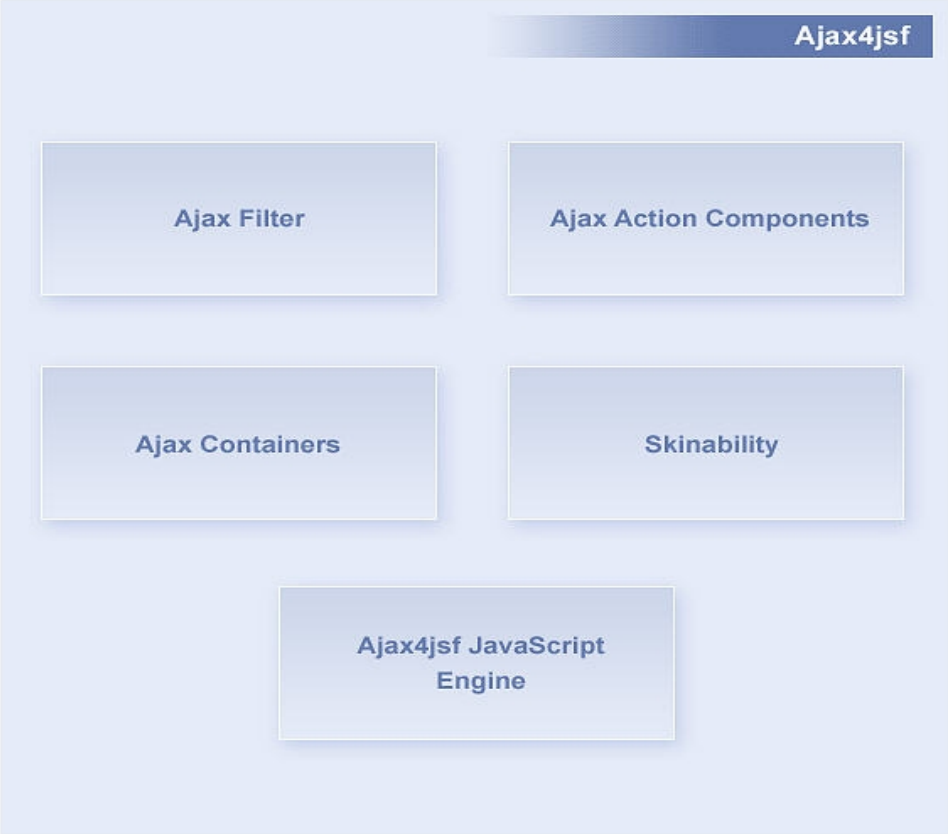
Az Ajax4jsf különbözik más AJAX megközelítésektől abban, hogy megengedi az oldal-orientált AJAX támogatást is a hagyományos komponens-orientált támogatással szemben. Ez azt jelenti, hogy egy oldalon definiálhatunk egy eseményt, mely generál egy megfelelő AJAX kérést, majd definiálhatunk olyan területeket az oldalon, melyeknek a kérés szerveroldali feldolgozása után szinkronizáltnak kell lenniük a JSF komponensfával. Tehát nem csak egy komponenst, hanem az oldal egyes területeit generáljuk újra.

Az Ajax4jsf néhány erőssége:

- Erősíti a JSF előnyeit: Teljes mértékben integrált a JSF életciklusával *(Függelék III.2.2)*. Míg más keretrendszerek csak a kezelt *Java Bean*ekhez enged hozzáférést, az Ajax4jsf támogatja az *action* és *value change listeners*, szerveroldali validátorok és konverterek használatát egy AJAX *Response-Request* ciklus alatt.
- AJAX támogatást ad már elkészült JSF alkalmazásokhoz: A keretrendszer egy komponens-könyvtár használatával lett implementálva, mely AJAX funkcionalitást ad már létező oldalakhoz, így nincs szükség JavaScriptek írására, vagy a létező komponensek lecserélésére.
- Lehetőséget ad beépített AJAX-támogatással rendelkező saját komponensek létrehozására. A közeljövőben kibocsátásra kerül egy teljes *Component Development Kit* (Komponens Fejlesztő Csomag – CDK) a saját Ajax4jsf komponensek egyszerű fejlesztésére.
- Különböző erőforrások kezelésének támogatása, mint képek, JavaScript kód, CSS stíluslapok. Az erőforrás eszközrendszer lehetővé teszi ezen erőforrások Jar fájlokba való csomagolását a saját komponensek kódjaival együtt.
- Az erőforrás eszközrendszer segítségével *on-the-fly* generálhatunk képeket; lehetővé teszi képek létrehozását *JavaGraphics2D* könyvtár használatával.
- Modern felhasználói interfész készítése *skin*-alapú technológiával: Különböző színsémák és egyéb UI jellemzők készítése és kezelése névvel hivatkozható *skin*-paramétereken keresztül.

Az Ajax4jsf támogatja a JSF specifikációjában szereplő összes *tag*et (komponenst). Az Ajax4jsf eszközeinek használatához csak annyit kell tennünk, hogy az Ajax4jsf könyvtárakat elhelyezzük a project *lib* könyvtárában, valamint a *faces-config.xml* konfigurációs fájlhoz a megfelelő *filter-mapping*-et hozzá kell adnunk.
A *web.xml* webalkalmazás deszkriptorban több inicializációs paramétert is megadhatunk, melyekkel az Ajax4jsf kezdeti konfigurációját állíthatjuk be.

Architektúra:



Ajax Filter: Ahhoz, hogy az Ajax4jsf eszközei elérhetőek legyenek egy alkalmazás számára, regisztrálnunk kell egy AJAX filtert a *web.xml* állományban. A filter különböző *Request* típusokat határoz meg. *(Függelék III.2.1)*
Ajax Action Components: Négy különböző AJAX *Action Component* áll rendelkezésünkre: *AjaxCommandButton, AjaxCommandLink, AjaxPoll* és *AjaxSupport*, melyek AJAX kérések küldésére szolgálnak.
Ajax Containers: Egy AJAX kérés küldése során újrarenderelendő területet meghatározó interfész egy JSF oldalon. Az *AjaxViewRoot* és *AjaxRegion* implementálják ezt az interfészt.
JavaScript Engine: Kliensoldali JavaScript motor

IV. PostgreSQL

A PostgreSQL egy objektumrelációs adatbáziskezelő-rendszer (ORDBMS), melyet a Kaliforniai Egyetem Berkeley Számítógéptudományi Tanszéke által fejlesztett POSTGRES 4.2-es verziója alapján készítettek.

A PostgreSQL egy *open-source* utódja az eredeti Berkeley implementációnak. A standard SQL legnagyobb hányadát támogatja, emellett több modern szolgáltatást is nyújt:

- komplex lekérdezések
- külső kulcs
- triggerek
- nézetek
- tranzakció-integritás
- többváltozatú konkurrenciavezérlés

Továbbá kiterjeszthető a felhasználó által például saját:

- adattípusok
- függvények
- operátorok
- aggregált függvények
- indexmetódusok
- procedurális nyelvek

definiálásával.

Open-source mivolta okán bárki által korlátlanul felhasználható, módosítható, terjeszthető akár saját, kereskedelmi vagy oktatási célokra is.

Architektúra:

A PostgreSQL kliens-szerver modell alapján működik, melyben egy munkafolyamat az alábbi processzekből áll:

- Egy szerverfolyamat, amely az adatbázis állományait kezeli, adatbáziskapcsolatokat fogad egy kliensalkalmazástól, és műveleteket végez az adatbázison a kliens kérésének megfelelően.
- Egy kliensalkalmazás, melyen keresztül a felhasználó műveleteket szeretne végezni az adatbázison. A kliensalkalmazások rendkívül sokfélék lehetnek: karakteres vagy grafikus eszköz, egy webszerver, amely weboldalait az adatbázis tartalma alapján jeleníti meg, vagy

éppen egy speciális adatbáziskezelő alkalmazás, mely az adatbázis könnyű karbantartását segíti elő. A PostgreSQL kiadvány tartalmaz néhány kliensalkalmazást, de legtöbbjüket a felhasználók fejlesztik.

Mint ahogy az általában egy kliens-szerver alkalmazásra jellemző, a kliens és szerver lehet külön munkaállomáson. Ebben az esetben a kommunikáció TCP/IP kapcsolaton keresztül történik.

A PostgreSQL szerver képes konkurrens kapcsolatot kezelni, minden egyes kapcsolathoz egy új folyamatot épít fel, mely a szerverfolyamattól függetlenül működik.

Néhány fontos jellemző:

- Támogatja a *Large Object*-ek (LOB) létrehozását és kezelését: adatfolyamjellegű hozzáférést biztosít a nagyméretű adatokhoz. Minden LOB a `pg_largeobject` rendszertáblában kerül elhelyezésre, méretük maximum 2 GB lehet. LOB-ok kezelésére rendelkezésre bocsát egy API-t, mely a nagyméretű objektumok írására és olvasására szolgál.
- Létrehozhatunk saját típusokat: több beépített egyszerű és összetett típus áll rendelkezésünkre, emellett lehetőség van saját típusok létrehozására is, melyek szintén lehetnek egyszerűek és összetettek.
  Kompozit típus: típussal rendelkező attribútumokat tartalmaz. Ily módon létrehozott típus függvény paramétere, visszatérési értéke vagy adatbázistábla oszlopának típusa esetleg újabb típus egy attribútumának típusa lehet.
  Skalár típus: a típus létrehozása előtt két vagy több függvényt kell létrehoznunk. Egy `input_function` a típus külső sztring reprezentációját konvertálja a típushoz tartozó operátorok és függvények által használható belső reprezentációra; visszatérési típusa az újonnan létrehozott típussal egyező kell legyen. Egy `output_function` az előbbi művelet inverzét végzi, visszatérési típusa `cstring`, amely beépített pszeudó-típus; C sztringet reprezentál. A függvényeket kötelezően C vagy hasonló alacsonyabb szintű nyelven kell megírnunk.
- Rendelkezésre bocsát egy tömb (`Array`) típust: létrehozhatunk többdimenziós tömböt, mely adatbázistábla oszlopának típusa is lehet. Viszont a tömbelemek csak egyszerű típusúak lehetnek. Rendelkezésre áll néhány beépített függvény, melyek a tömbök manipulálsát szolgálják, ezenkívül a sztring és `array` közötti konverzió biztosított.
- Öröklődés: az adatbázistáblák között létezik egyfajta öröklődés, mely akár többszörös is lehet. Egy újonnan létrehozott adatbázistábla örökölheti más már létező táblák oszlopait, ezzel együtt sorait is.

- Pszeudó-típusok: A PostgreSQL rendszer tartalmaz néhány speciális típust, melyekkel nem deklarálhatók adatbázistáblák oszlopai, viszont függvények visszatérési típusa vagy paramétereinek típusa lehet. Használatuk akkor előnyös, ha egy függvény működése nemcsak egy-egy egyszerű SQL típushoz köthető. Magasabb absztrakciós szintet biztosít. Pszeudó-típus például az `any` (bármilyen típus), `anyarray` (bármilyen típusú tömb), `record` (határozatlan sortípus) stb.
- Több procedurális nyelv használatát támogatja: PL/pgSQL, PL/Tcl, PL/Perl, PL/Python nyelveken írhatunk függvényeket a C mellett.

V. Apache Tomcat

Az Apache Tomcat egy *Servlet/JSP Container*, mely a *JavaCommunityProcess* által kialakított JSP és Servlet specifikációkat implementálja. Futtató környezetet biztosít az előbbi technológiák alapján fejlesztett webalkalmazások és webszolgáltatások számára. Elérhető Windows és Unix alapú platformokon, ingyenesen letölthető a http://tomcat.apache.org weboldalról.
Néhány fontosabb jellemző:

- *Apache Tomcat Deployer*: egyfajta installációs folyamat (*deploying*) elvégzését teszi lehetővé. A *deploy* befejeztével adott webalkalmazás elérhetővé válik a Tomcat számára.
- *Manager*: Beépített webalkalmazás, mely a *deploy, undeploy, redeploy* műveletek elvégzését könnyíti meg a felhasználó számára.
- Regisztrálhatunk JNDI és JDBC erőforrásokat, melyeket elérhetnek a futó webalkalmazások
- Összetett, jól konfigurálható *logging management*

VI. Technológiai áttekintés

Az előzőekben tárgyalt technológiák használhatóságának és integrálhatóságának vizsgálata céljából egy webalkalmazást készítettem az alábbiaknak megfelelően:

- Servlet container: Apache-Tomcat-5.5.20
- Adatbáziskezelő-rendszer: PostgreSQL-8.2.3
- Megjelenítés:
  - JSF-1.1
  - Ajax4jsf-1.1.0
  - Tomahawk-1.1.3 (Apache által implementált JSF extension)
  - CSS
- Perzisztenciakezelés: Hibernate-3.2.0 (core + annotations)
- IDE: eclipse-SDK-3.2.2
- Célzott böngésző: Mozilla-Firefox
- A fejlesztés alatt használt operációs rendszer: Gentoo-Linux

Az alkalmazás elkészítésének menete:

1. Feladat megfogalmazása, követelmények kialakítása *(Függelék VI.1)*
2. Technológiák, eszközrendszerek kiválasztása *(Függelék VI.2)*
3. Koncepcionális terv kialakítása *(Függelék VI.3, Függelék VI.4, Függelék VI.5)*
4. Adatbázisséma megtervezése *(Függelék VI.6)*, kialakítása, az adatbázis feltöltése
5. A JSF oldalak, a szükséges perzisztens objektumok példányosító osztályainak létrehozása, a logikai réteg implementálása párhuzamosan
6. Tesztelés

Az adatbázisterv kialakítása során szem előtt tartottam a bővítés és továbbfejlesztés lehetőségét, ezért olyan táblákat és oszlopokat is létrehoztam, melyek az alkalmazás aktuális követelményein túlmutatnak. Ettől függetlenül az adatbázis kerek egészt alkot, emellett könnyedén integrálható egy tanulmányi rendszert megvalósító bővebb alkalmazásba is. Az adatbázisterv nem tartalmazza a nézeteket, triggereket, tárolt eljárásokat, ugyanis ezeket az alkalmazás fejlesztése alatt az éppen aktuális elvárásoknak megfelelően alakítottam ki.

Célom volt, hogy a lehető legtöbb adatmanipulációs és ellenőrző funkciót az adatbázis szintjére vigyem. Ebben segítségemre volt több a PostgreSQL által nyújtott lehetőség, mint triggerek és tárolt eljárások létrehozása. Ezek implementálására a PL/pgSQL procedurális nyelvet használtam. Sajnos a nyelvnek igen sok hiányossága mutatkozott az Oracle PL/SQL-jéhez viszonyítva; ez nagyban hátráltatta munkámat. Az együttes hozzárendelés, a kollekciók létrehozásának és kezelésének hiánya, a szegényes kivetelkezelés mind-mind a PL/pgSQL ellen szól.

További hiányosságok merültek fel a PostgreSQL mint adatbáziskezelő-rendszer esetében. Az előre elgondolt adatbázisséma teljes átgondolását kényszerítette ki az, hogy az `Array`-t leszámítva nincsenek kollekciótípusok. Viszont az `Array` nem tartalmazhat komplex típusú elemeket, így eleve kizárt volt objektum-relációs elképzeléseim nagy része. Sok esetben jóval egyszerűbb lett volna kollekció típusú oszlopok létrehozása ahelyett, hogy a relációs modellnek megfelelően külső kulcs segítségével valósítsak meg tartalmazási viszonyt. Ezen akadály leküzdésére az előbbi módszert (külső kulcs) alkalmaztam, valamint néhány esetben a sorok számának csökkentése érdekében egy-egy a külső kulcsokat tartalmazó `Array`-t alakítottam ki a megfelelő táblában, így szimulálva egy-egy 1-N kapcsolatot. Ennek hátránya azonban az integritási megszorítások nehézkes ellenőrzése – mely innentől kezdve a rendszer által nem biztosított –, valamint ennek következményeként a konzisztencia fenntartása is kevésbé könnyed.
Továbbá az Oracle-höz szokott fejlesztőknek némi bosszúságot okozhat, hogy a PostgreSQL külső összekapcsolások esetén csak az ANSI szabványnak megfelelő JOIN-okat támogatja. Mindezek ellenére a PostgreSQL adatbáziskezelő-rendszer használhatónak bizonyult.

A fejlesztést a használt eszközök, technológiák konfigurálásával folytattam. A könnyű módosítás és hordozhatóság biztosítása érdekében az adatbáziskapcsolatot minden esetben JNDI néven keresztül kérem a servlet containertől, ezért első lépés mindig egy *DataSource* definiálása a Tomcatben. Ez megvalósítható globális adatforrás definiálásával a Tomcat által biztosított adminisztrációs felületen, vagy a Tomcat konfigurációs (XML) fájljain keresztül. Magasabb fokú hordozhatóságot biztosít, ha az alkalmazást tartalmazó archív állományban egy alkalmazásspecifikus konfigurációs (XML) fájlt hozunk létre, melyet a Tomcat a *deploy* során felhasznál, és ennek megfelelően alakítja ki az erőforrásokat. Éppen ezért ezt a lehetőséget választottam.

Következő lépésben a Hibernate beállítását végeztem el. Konfigurációs állományként a `hibernate.cfg.xml`-t választottam, mely a Hibernate Annotations működtetéséhez és beállításához tökéletesen megfelel. Itt állítottam be a használt dialektust (`org.hibernate.dialect.PostgreSQLDialect`), a JNDI D*ataSource* mint erőforrás

használatát, valamint a perzisztens objektumokat példányosító osztályokat. Így az `AnnotationConfiguration` példányosításán kívül további – programkódból történő – konfigurációra nem volt szükség. Ennek elérése az utólagos módosítások egyszerűsítése végett fontos.

Következő teendőm a JSF és kiterjesztéseinek beállítása volt, mely a `faces-config.xml` konfigurációs fájl megfelelő kialakításából áll. Ez viszonylag egyszerű művelet, a J2EE tutorial és a kiterjesztések hivatalos honlapja megfelelő dokumentációt nyújt. Az alkalmazást felkészítettem egy lehetséges többnyelvűsítésre, ez némi járulékos, de egyszerű konfigurációval járt; a JSF megfelelő, könnyen használható eszközöket nyújt a többnyelvű alkalmazások fejlesztéséhez..

A beállítások elvégzése után következhetett a konkrét implementáció elkészítése. A perzisztens osztályokat és a JSF *backing bean*jeit párhuzamosan alakítottam ki, egy-egy újabb weboldal és/vagy funkció hozzáadása során. A Hibernate és a JSF működésének összehangolása könnyű abból a szempontból, hogy a perzisztens osztályok és a *backing bean*ek könnyen megfeleltethetők egymásnak. Az egyszerűség kedvéért létrehoztam egy olyan segédosztályt, mely az *entity* és JSF-*bean*ek attribútumainak másolását végzi; erre a *JavaBeans* technológia eszközeit használtam. Így egy újonnan létrejött JSF-bean adattagjainak másolásával könnyedén elvégezhető egy perzisztens objektum adattagjainak beállítása, és fordítva.
Az adatbázisműveletek során keletkező kivételek (pl. hibás azonosító vagy jelszó megadása a bejelentkezés során – *Függelék VI.9*), hibák (pl. nem elérhető adatbázis), validáció (*Függelék VI.10*) üzenetei is könnyedén megjeleníthetők a weboldalakon. A hibaüzenetek regisztrálására egy újabb segédosztályt hoztam létre. Ennek feladatkörébe tartozik a `FacesContext` aktuális példányának lekérése és az alkalmazás által generált hibaüzenetek vagy információk regisztrálása egy-egy oldalon. A fent említett segédosztályok feladatkörének meghatározásában és funkcióinak kialakításában sok segítséget nyújtott a
http://www.oracle.com/technology/pub/articles/cioroianu_jsfdb.html weboldalon található *Building Database-driven applications with JSF* című leírás.

Gondot jelenthet a Hibernate munkafolyamatainak JSF-fel szinkron kezelése. A Hibernate hivatalos weboldalán találhatunk több a Hibernate architekturális és tranzakciókezelésére vonatkozó tervezési mintát. Ezen kívül a weben több használható – a JSF és Hibernate összehangolt működését támogató – tervezési minta is megtalálható. A legelterjedtebb talán az a módszer, melyben minden egyes HTTP *Request*-hez létrehozunk egy új *session*-t. Ezt könnyen automatizálhatjuk olyan *Filter Servlet* vagy *Phase Listener* implementálásával, melyek a Hibernate munkafolyamatait kezelik. Ennek a módszernek a hátránya, hogy nem tudunk több *Request*-en

átívelő Hibernate *session*-t kezelni, mely bizonyos esetekben elvárás lehet. A JSF és Hibernate összehangolását egy olyan segédosztály létrehozásával oldottam meg, amely az adatok tárolását és lekérdezését végzi, és ennek megfelelően kezeli a munkafolyamatokat, tranzakciókat. Így egy *session*-t csak akkor nyit meg és zár le, ha az indokolt. Annak érdekében, hogy az osztály definíciója általános legyen, a lekérdezéseket és DML utasításokat a JSF-beanek hozzák létre, és adják paraméterül a megfelelő metódusoknak.
A perzisztens objektumok kezelésére a Hibernate *session-per-request* stratégiáját használtam. A bonyolultabb lekérdezéseket az adatbázis szintjére vittem nézetek létrehozásával (a Hibernate szemszögéből egy nézet nem különbözik egy adatbázistáblától). Így ez az egyszerű tervezési minta is elegendőnek bizonyult.
A megfelelő tranzakciókezelési stratégia kiválasztásától eltekintve a JSF és a Hibernate működése egyszerűen összehangolható.

Mivel az alkalmazás gerincét a JSF képviseli, a legtöbb probléma az ehhez kapcsolódó implementációs lépések során adódott. A JSF általam használt verziója (JSF-1.1) még viszonylag kiforratlan abból a szempontból, hogy kevés a standard implementáció által biztosított komponens áll rendelkezésre. Komoly hátránynak tartom például a HTML `<div>` elemét reprezentáló komponens hiányát. Hozzáteszem, hogy a JSF-1.2-ben már elérhető olyan eszköz, mely segítségével `<div>` jeleníthető meg, viszont ennek implementációja a JSP-2.0 specifikációt használja, mely az általam alkalmazott Tomcat-5.5 által még nem támogatott.
Korábbi munkáim során szembesültem a fájlfeltöltés nehézségeivel, ugyanis nem áll rendelkezésre az ezt megvalósító komponens. Viszont szerencsére mindkét problémát megoldja az Apache által fejlesztett Tomahawk JSF kiterjesztés. Ezért a standard JSF implementáció mellé ezt is használtam.

Nagy fejtörést okozott `<CommandButton>`-ok egy `<DataTable>` soraiban való megjelenítése. Ugyanis látszólag működött, a Tomcat semmiféle hibát nem regisztrált, viszont egyik `<ComandButton>` sem funkcionált a megfelelő módon, konkrétan a mögöttes *action* metódus nem került meghívásra. A probléma megoldására – mint nagyon sok esetben – különböző szakmai fórumok derítettek fényt. Néhány órás böngészés után kiderült, hogy a *Request-scope*-pal rendelkező objektumok esetén a `<CommandButton>`-ok azonosítói – melyek a rendeltetésszerű működéshez elengedhetetlenek – összekeverednek a `<DataTable>`-ben. Ebből kifolyólag a `<CommandButton>`-ok használhatatlanná válnak. A problémát a standard JSF helyett a Tomahawk implementáció által nyújtott `<DataTable>` használata oldotta meg.

További fejtöréseket okozott az Ajax4jsf megfelelő használata. A Jboss által prezentált hivatalos dokumentáció nagyon jó architekturális áttekintést nyújt, és megfelelő támogatást biztosít a kezdeti lépések megtételéhez, ha egy JSF alkalmazáshoz AJAX támogatást szeretnénk hozzáadni. Viszont az összetettebb funkciók kialakítása során ez a segítség kevésnek bizonyult. Legnagyobb problémát az okozta, hogy a tutorial szerint elegendő a `<support>` *tag*-nek megadni egy AJAX *request* során újrarenderelendő komponens azonosítóját. Ez olykor elegendő, és a megfelelő funkcionalitást nyújtja. Viszont bizonyos esetekben kevés, mert nem állítja be a mögöttes *backing bean* adott attribútumát. Látszólag működőképes, viszont némi tesztelés után érdekes helyzetek alakulnak ki; egy-egy AJAX *request*-et indukáló tevékenység megismétlése a probléma fennállására fényt derít. Viszont mivel hibaüzenet, kivétel vagy bármilyen egyéb jelzés nem generálódik, a megoldás kialakítása valamint a hiba pontos lokalizálása igen nehézkes. Végül a Mozilla-Firefox Firebug kiegészítésének használata *debug*-olás céljából, továbbá fórumok hosszas böngészése segített a probléma kiküszöbölésében: a megoldást egy `<AjaxRegion>` definiálása nyújtotta a kérdéses oldalterületen, melynek hatására a háttérben levő *backing bean* megfelelő attribútuma(i) kötelezően beállításra kerülnek. Végső következtetésem az, hogy az adattagok határozatlan módon, bizonyos esetben beállítódtak, máskor viszont nem, tehát a JSF *Update Model Values Phase* fázisában szereplő viselkedés kikényszerítése volt a cél. Ennek tudatában már a hivatalos dokumentáció is megfelelő segítséget nyújt, hiszen szót ejt az *Ajax container* interfészről, mely a kötelezően dekódolandó oldalterületek definiálására szolgál.
A kérdéses weboldalak a *Függelék VI.7* és *VI.8* pontja alatt láthatók. Mindkét oldalon egy listából választhatunk értékeket, melynek megfelelően az oldal tartalma megváltozik. Első esetben ha a hallgató több szakot hallgat párhuzamosan, a listából kiválaszthatja azt a szakot, melynek adatait látni szeretné. A lista aktuális értékének megváltoztatása egy AJAX *request*et generál, mely az új értéknek megfelelő adatokat fogja kérni az alkalmazáson keresztül az adatbázistól. Az AJAX-nak köszönhetően az oldalnak csak azon részét kell újrarenderelni, mely az aktuálisan kiválasztott szakra vonatkozó adatokat tartalmazza. A második esetben az egy-egy szemeszterre vonatkozó információk jeleníthetők meg a lista aktuális értékének változtatásával.

Úgy gondolom, hogy a kiegészítések integrálása a standard JSF implementáció mellé nagyon egyszerűen végrehajtható. A legkevesebb fejtörést, és a legegyszerűbb megoldásokat a Tomahawk-extension nyújtotta. Az Ajax4jsf annál több problémát okozott, viszont a kezdeti nehézségek leküzdése után megfelelően működött, nem beszélve arról, hogy nem kellett végigjárnom az AJAX támogatás hozzáadásának hagyományos JSF-útját, mely véleményem szerint igen időigényes, és nagyon sok problémát felvet.

A JSF oldalakon lévő komponensek megfelelő megjelenítését CSS használatával alakítottam

ki. A CSS által nyújtott eszközök a JSF komponenseivel igen hatékonyan kombinálhatók.

A Tomcat servlet container beállításával, kezelésével sok problémám nem adódott. Segítségével egy webalkalmazás működtetéséhez szükséges teendők egyszerűen és hatékonyan elvégezhetők.

Az eddigiek alapján úgy vélem, hogy az előző részekben bemutatott technológiák, eszközök, eszközrendszerek alkalmasak egy modern webalkalmazás elkészítésére, egymással való kompatibilitásuk elfogadható.

VII. Összefoglalás

Az előző fejezetekben tárgyalt technológiák rendkívül összetettek. Nagyon sokrétű szolgáltatásokat nyújtanak, magas absztrakciós szintet képviselnek, jól konfigurálhatóak. Éppen ezért megfelelő használatuk mély technológiai ismereteket és igen sok tapasztalatot igényel mind a tervezést, mind a megvalósítást tekintve. Az egyes technológiákhoz kapcsolódó tervezési minták megismerése, felhasználása elengedhetetlen egy jól működő alkalmazás megtervezéséhez és implementálásához, hiszen alapvető elvárásnak kellene lennie annak, hogy egy szoftver ne csak működjön, hanem jól működjön, hatékony, gyors, skálázható, könnyen karbantartható és továbbfejleszthető legyen.

Annak ellenére, hogy a fejlesztés során az egyes feladatok viszonylag könnyen elkülöníthetőek, köszönhetően a jól felépített technológiáknak és a rétegzettségnek, mégis már egy kisebb alkalmazás is könnyen áttekinthetetlenné válhat egy fejlesztő számára. Egy közepes méretű alkalmazás elkészítése pedig mindenképpen legalább néhány fős fejlesztői csapatot igényel véleményem szerint.

Természetesen az előzőekben tárgyaltak mellett nagyon sok egyéb technológia, eszközrendszer is rendelkezésünkre áll egy webalkalmazás fejlesztése során. Az Oracle mint adatbáziskezelő-rendszer egy jó alternatíva lehet az általam bemutatott PostgreSQL-lel szemben. Az adatbázis megtervezése, létrehozása, karbantartása egyszerűbb lehet az Oracle sokrétű objektumrelációs eszközei által. Az üzleti logika rendkívül nagy hányada levihető adatbázisszintre hatékonyságcsökkenés nélkül. Mindemellett a Hibernate nyújt olyan eszközrendszert, mellyel akár a legprimitívebb adatbáziskezelő-rendszer használata mellett is komoly üzleti logikát vihetünk szinte majdnem az adatbázis szintjére.

A JSF a kiegészítéseivel együtt már jól használható keretrendszert biztosít a megjelenítéshez. Előnyt jelenthet az is, hogy akár HTML-t, akár JSP-t is ágyazhatunk a JSF oldalakba kisebb-nagyobb nehézségek árán. Továbbá a JSF integrálható több ma használt keretrendszerrel (például Struts) egyaránt.

Az AJAX támogatás a mai webvilágban szinte nélkülözhetetlen; alapvető elvárássá vált egy webalkalmazással szemben. Az Ajax4jsf egyszerű megoldást nyújt ennek biztosítására. Természetesen hátrányként is értelmezhető az, hogy csak JSF-fel müködik.

Tapasztalataim alapján úgy vélem, hogy az ismertetett technológiák megfelelnek a mai modern elvárásoknak. Együttes alkalmazásuk hatékony és gyors fejlesztést tesz lehetővé, amennyiben a megfelelő ismeretek elsajátítása megtörtént; ennek hiányában viszont könnyen az *antipattern*-ek áldozatává válhatunk. Az említett eszközrendszerek kombinációjával komplex, robusztus alkalmazások készítése is lehetséges.

Irodalomjegyzék

J2EE tutorial:
http://java.sun.com/j2ee/1.4/docs/tutorial/doc/

Building Database-driven Applications with JSF (Andrei Cioroianu):
http://www.oracle.com/technology/pub/articles/cioroianu_jsfdb.html

Hibernate documentation:
http://www.hibernate.org/

Asynchronous JavaScript Technology and XML (Ajax) With the Java Platform (Greg Murray):
http://java.sun.com/developer/technicalArticles/J2EE/AJAX/

Ajax: A New Approach to Web Applications (Jesse James Garrett):
http://www.adaptivepath.com/publications/essays/archives/000385.php

AJAX Revolution:
http://www.telerik.com/community/ajax-learning-resources.aspx

JBoss Ajax4jsf:
http://labs.jboss.com/jbossajax4jsf/

PostgreSQL 8.2.4 Documentation:
http://www.postgresql.org

Apache Tomcat:
http://tomcat.apache.org/

Implementing Hibernate Open session per view in Apache MyFaces:
http://www.jroller.com/page/HazemBlog?entry=implementing_hibernate_open_session_per

Függelék

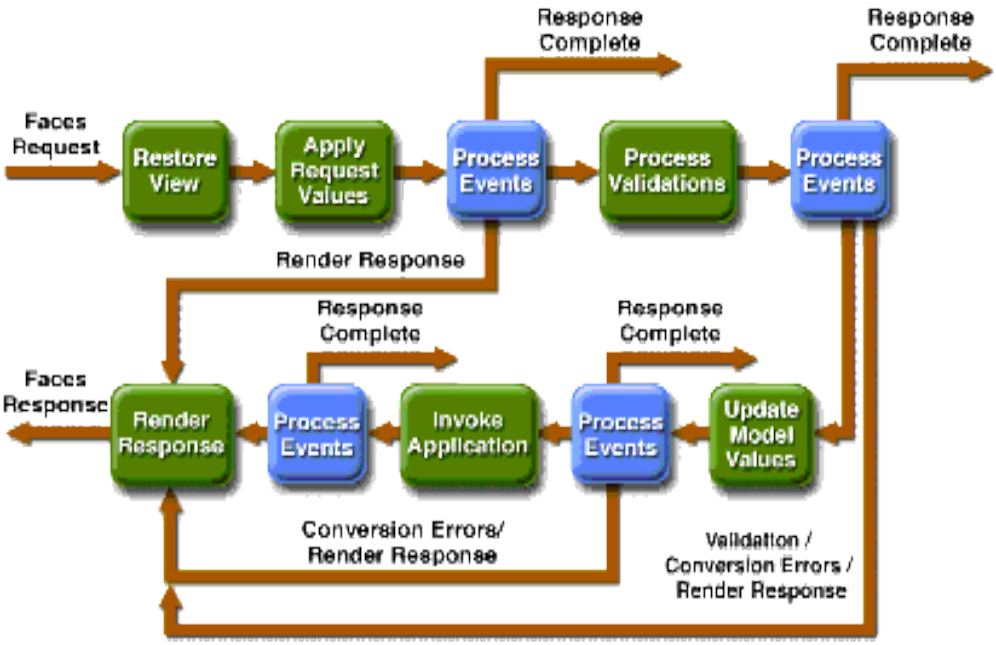
I.1 JSF *UICommand tag* kétféle módon jeleníthető meg:

Tag	Megjelenési forma
`commandButton`	Login
`commandLink`	hyperlink

I.2 Példa egyszerű JSF navigáció-konfigurációra:

```
<navigation-rule>
     <from-view-id>/greeting.jsp</from-view-id>
          <navigation-case>
               <from-outcome>success</from-outcome>
               <to-view-id>/response.jsp</to-view-id>
          </navigation-case>
</navigation-rule>
```

I.3 JSF *Request-Response* életciklus:



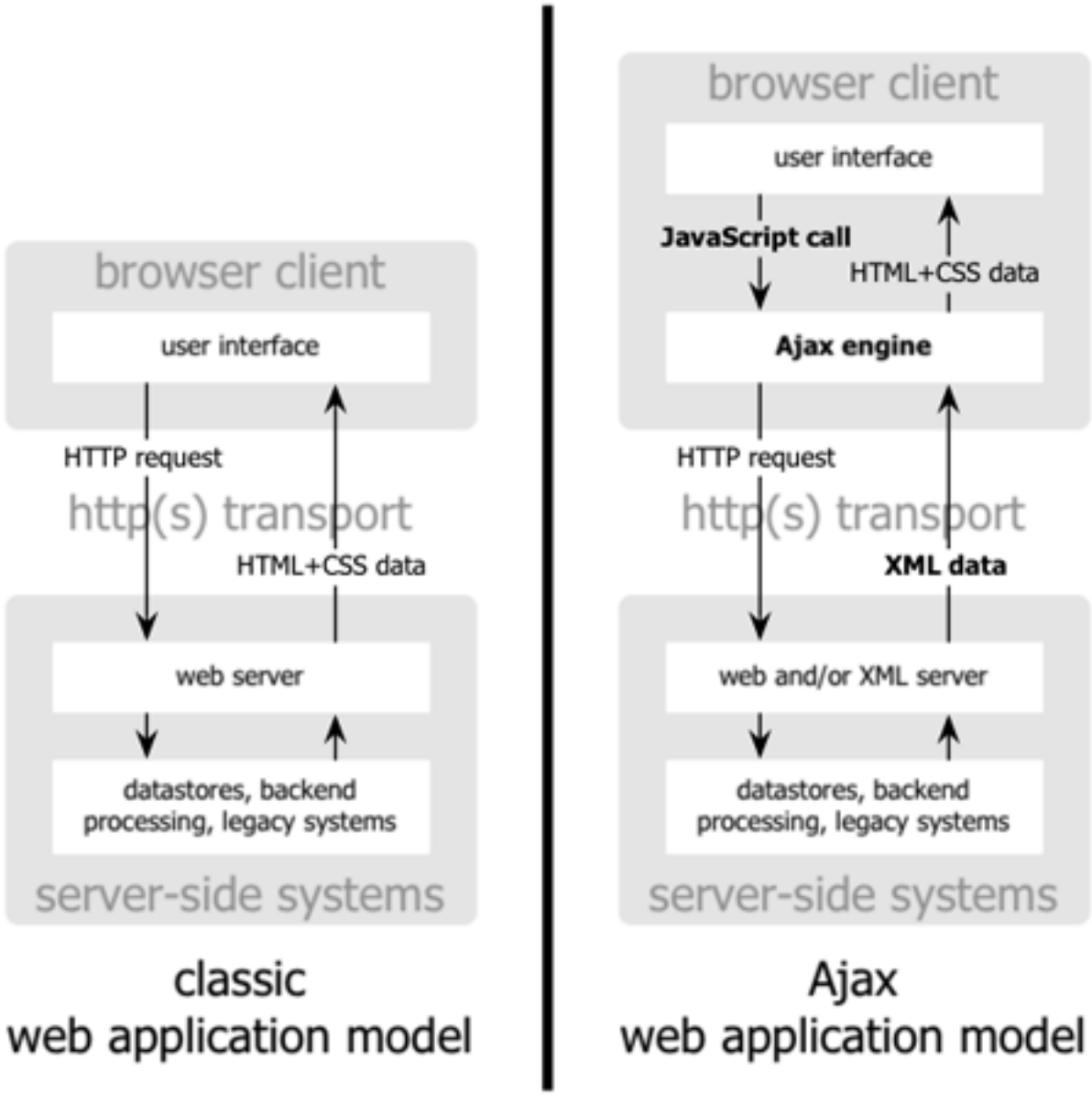
II.2.1 `AnnotationConfiguration` inicializálása:

```
import org.hibernate.*;
import org.hibernate.cfg.*;

public class HibernateUtil {

private static final SessionFactory sessionFactory;

    static {
        try {

            sessionFactory =
               new
     AnnotationConfiguration().buildSessionFactory();
        } catch (Throwable ex) {
            // Log exception!
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static Session getSession()
            throws HibernateException {
        return sessionFactory.openSession();
    }
}
```

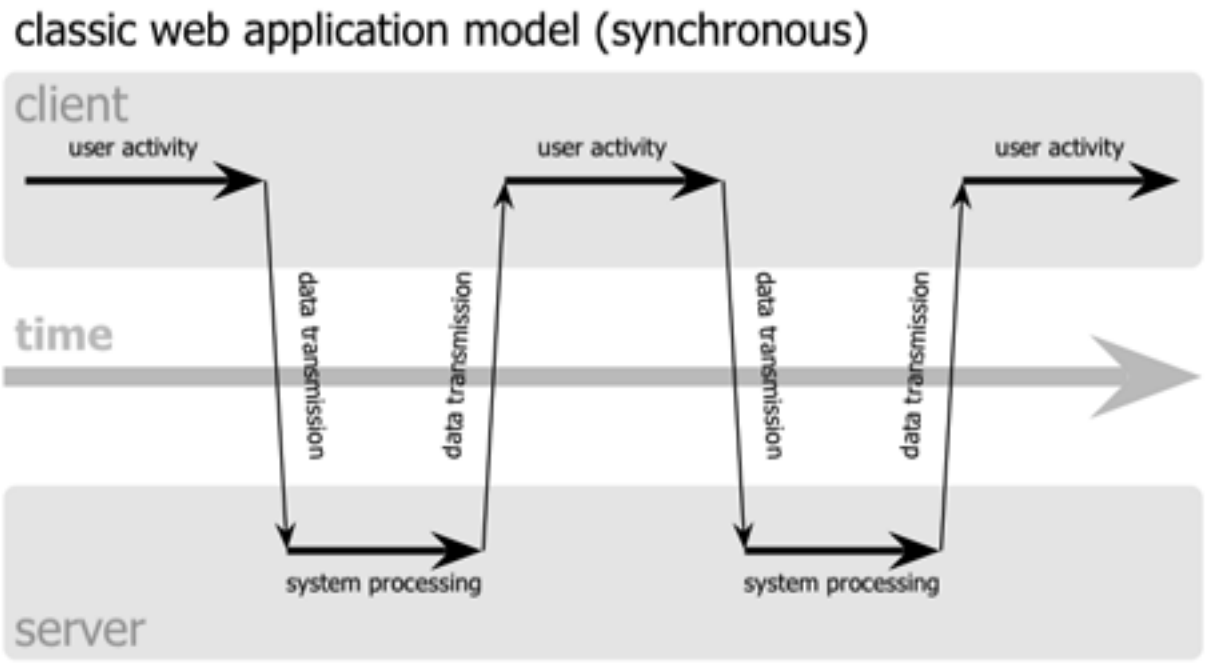
II.2.2 Példa egy `hibernate.cfg.xml` dokumentumra:

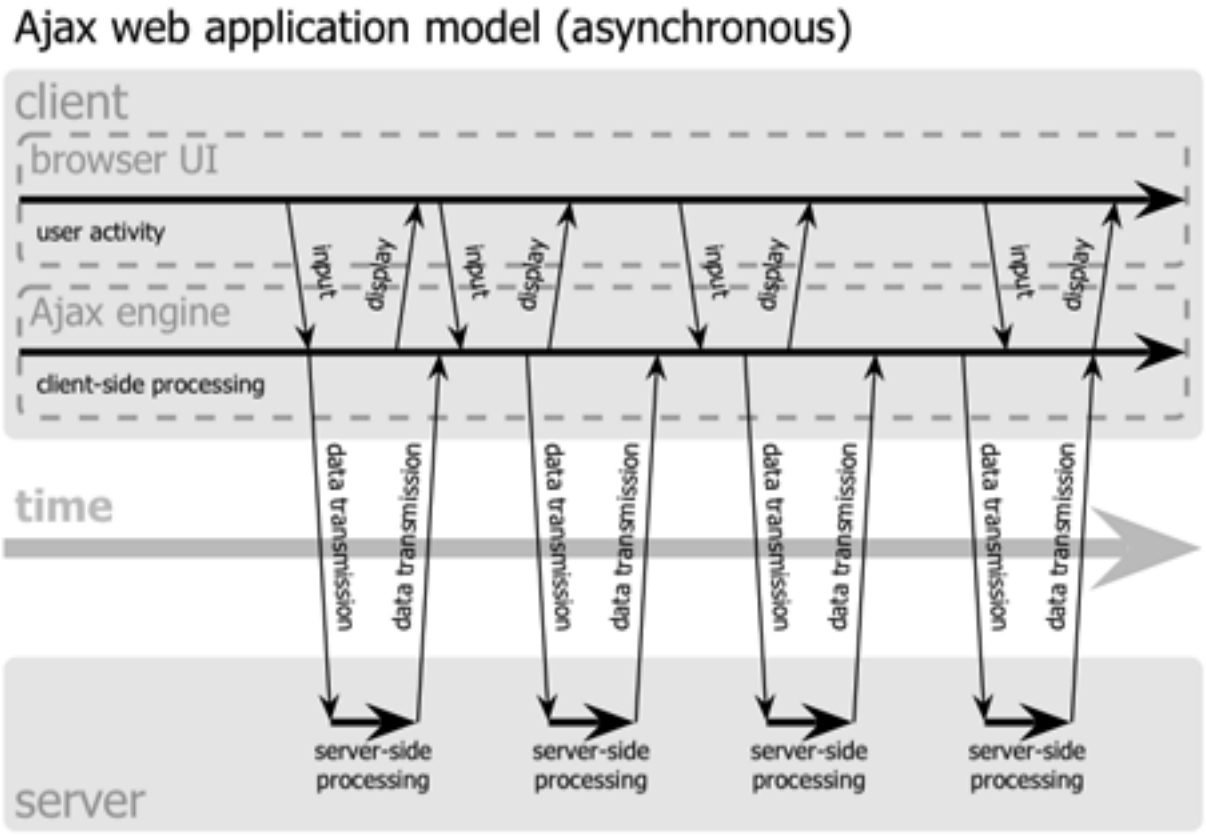
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
  "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-configuration
3.0.dtd">

<hibernate-configuration>
    <session-factory name="hib_session_factory">
          <property name = "hibernate.connection.driver_class">
               org.postgresql.Driver
          </property>
          <property name="hibernate.dialect">
               org.hibernate.dialect.PostgreSQLDialect
          </property>
          <property name="connection.datasource">
               java:comp/env/jdbc/a_database_name
          </property>
          <mapping class="model.AModelClass"/>
    </session-factory>
</hibernate-configuration>
```

III.1.1.a Klasszikus és AJAX webalkalmazás modell:

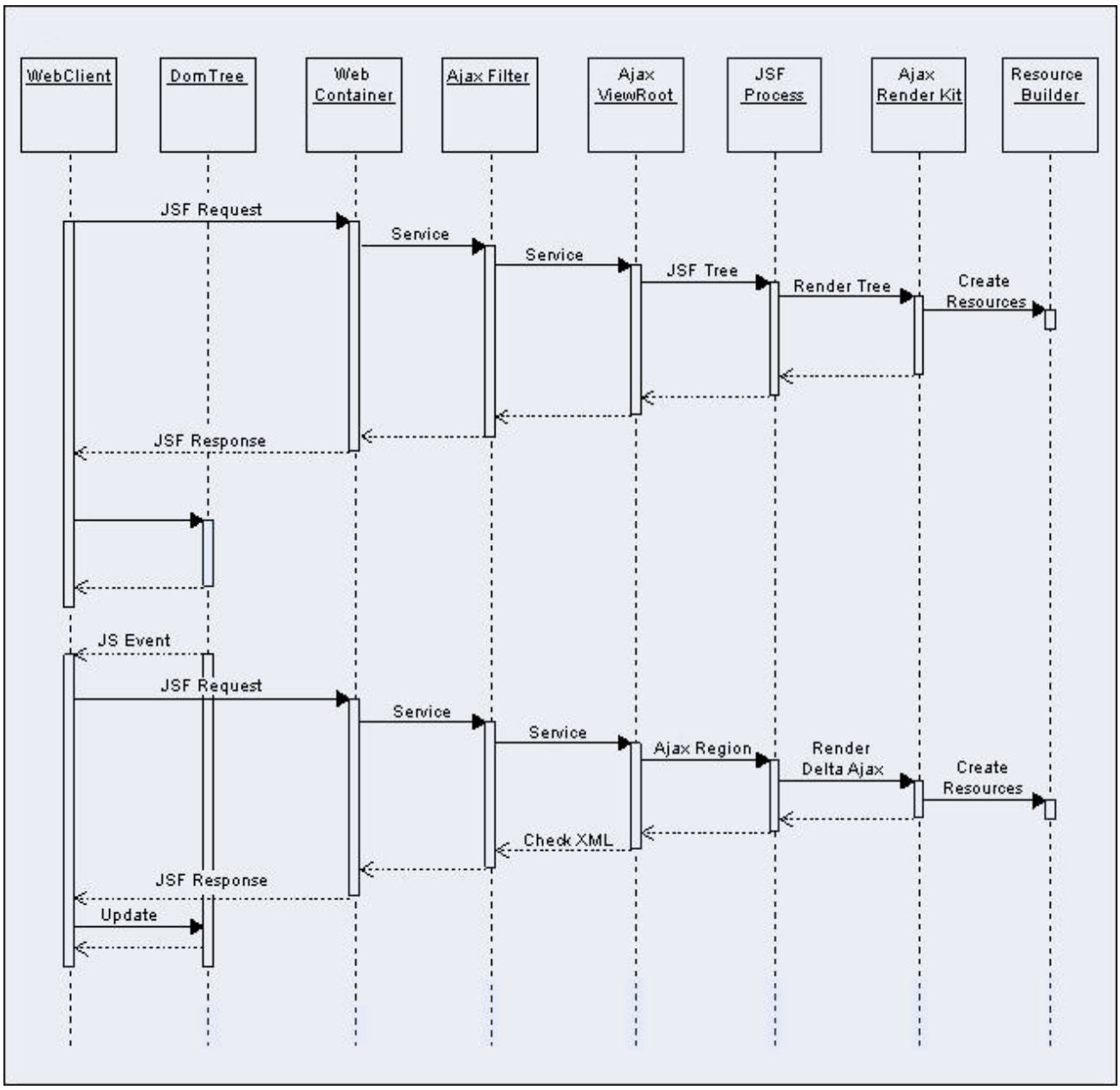


III.1.1.b Aszinkron interakció AJAX és szinkron interakció hagyományos módon:

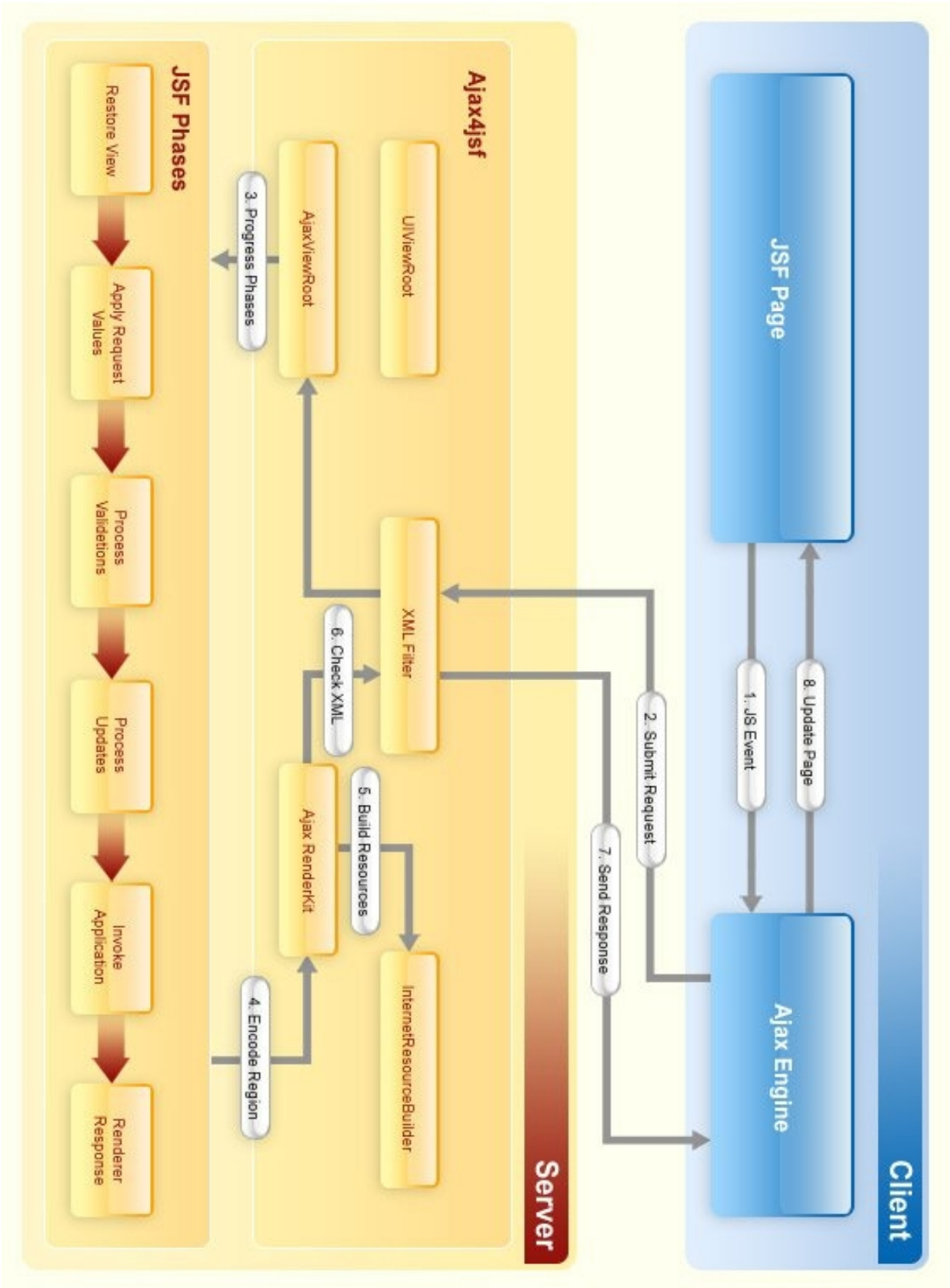




III.2.1 Ajax4jsf kérésfeldolgozás *sequence-diagram*ja:



III.3.2.2 Ajax4jsf életciklus:



VI.1 Feladat megfogalmazása, követelmények:

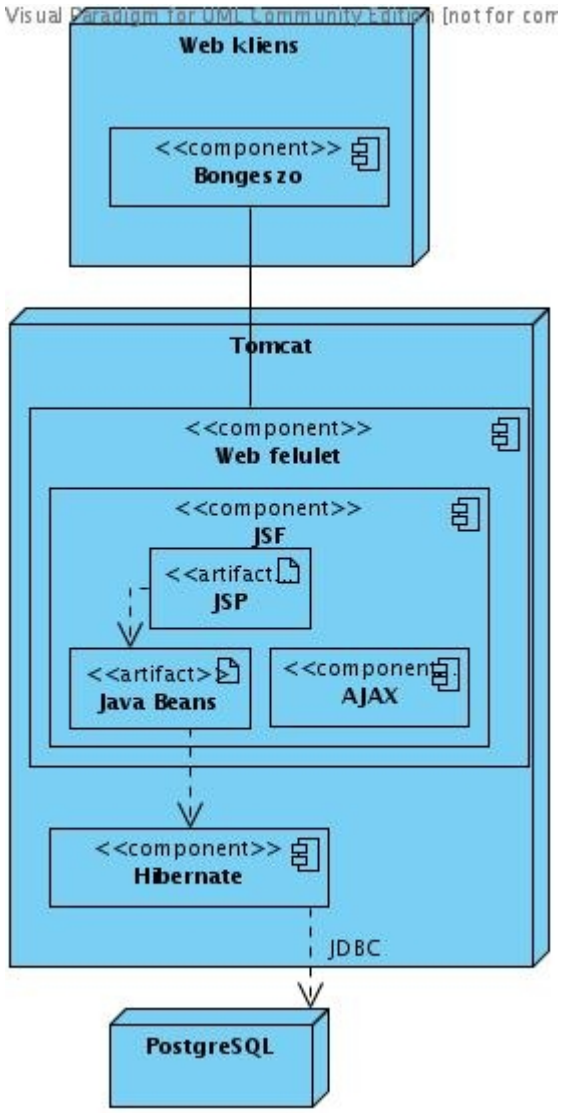
Feladat: Az alábbi követelményeknek megfelelő webalkalmazás fejlesztése

Take It Up – Tanulmányi rendszer

A Take It Up egy egyetemi tantárgyfelvételt lehető tevő (egyszerűsített) tanulmányi rendszer. A rendszerbe beépíthető a tantárgyi tematika, a szakok, a hallgatók, oktatók, a termek és az órarend (időpont, oktató, terem). A tantárgyi tematikában szereplő tárgyakat az oktatók meghirdethetik, és a követelményeknek megfelelő hallgatók felvehetik. A felvitt órarendnek konzisztensnek kell lennie a meghirdetett tárgyakkal.

A rendszer rögzíti a hallgatók előrehaladását, a különböző félévekben meghirdetett tárgyakat.

VI.2 Deployment diagram:



VI.3 Fogalomszótár:

Take It Up
Fogalomszótár

Felhasználó:
Olyan személy, akinek azonosítója és jelszava van a rendszerben. Személyes adatai tárolva vannak a rendszerben.

Oktató:
Olyan felhasználó, aki tárgyak menedzselését végezheti, tárgyakat hirdethet meg, vagy általa meghirdetett tárgyak meghirdetését vonhatja vissza.

Hallgató:
Olyan felhasználó, akinek adott szakra vonatkozó adatai vannak, az eddig eltöltött szemesztereinek megfelelően.

Adminisztrátor:
Új felhasználókat definiálhat, azokat módosíthatja vagy törölheti a rendszerből.
Az adatbázist nagyban érintő (tematika, órarend feltöltése) feladatokat látja el.

Index:
Egy hallgatóra vonatkozó minden információ innen érhető el.

Tárgy, szak, tematika, kurzus, tárgy kategória:
A Debreceni Egyetemnek megfelelően.

VI.4 Folyamatok:

Take It Up
Folyamatok

Belépés:
Az admin által regisztrált felhasználó megadja azonosítóját és jelszavát

Tárgy(ak) keresése:
Az oktató és a hallgató kereshet a tárgyak között. Megadhat kategóriát (adott szakon kötelező, sávos és modul tárgyak), hirdető kart, tárgynevet vagy tárgykódot.

Tárgy részleteinek megtekintése:
Adott tárgyat kiválasztva megtekinthetjük bővebb részleteit, az aktuális felhasználó jogosultságának(hallgató/oktató) megfelelő módosításokat elvégezhetjük a tárgyon.

Előrehaladás megtekintése:
A hallgató valamely szakját kiválasztva megtekintheti eddigi eredményeit, felvett tárgyait, átlagait a különböző félévekben.

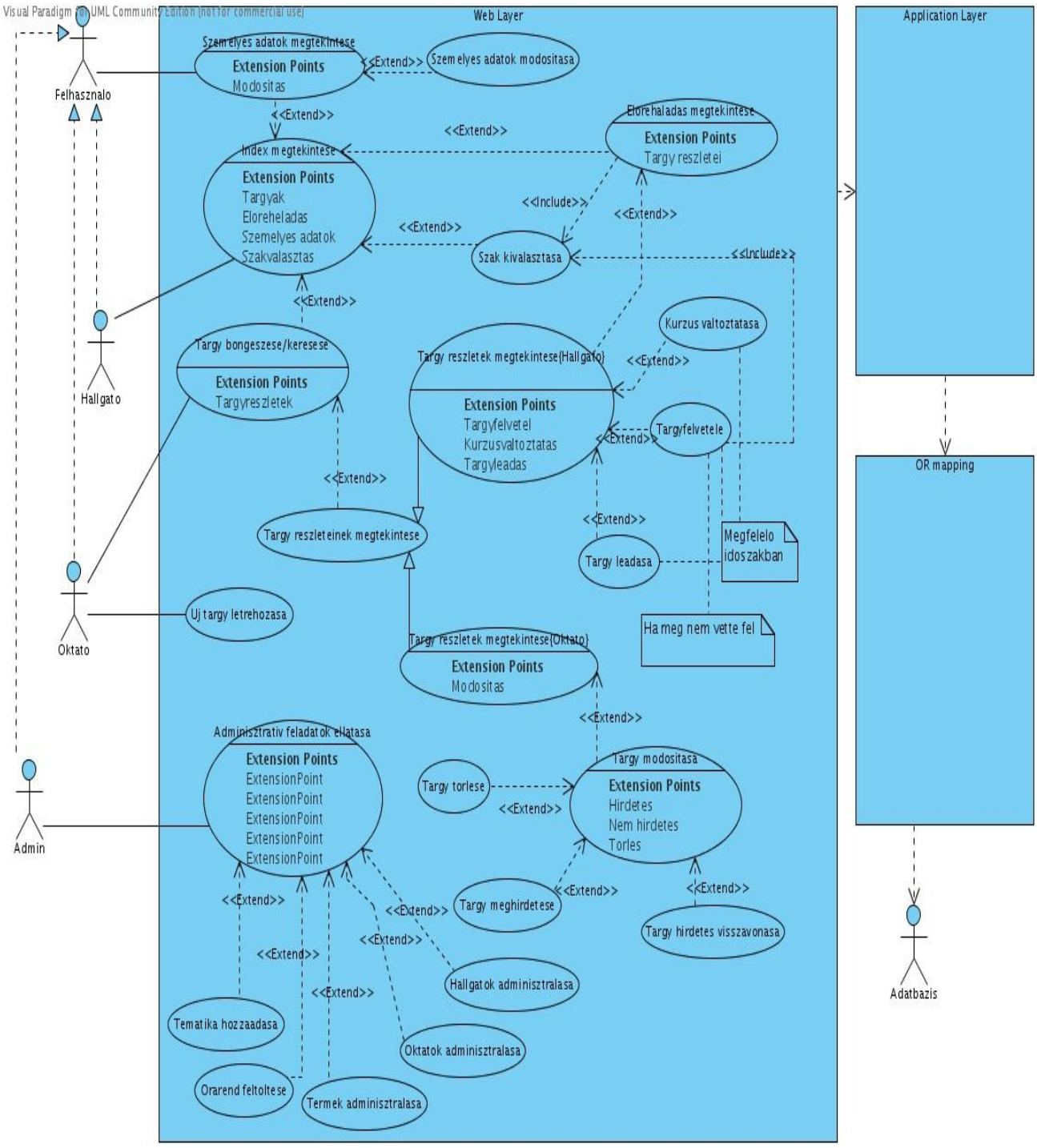
Tárgy felvétele:
Amennyiben a hallgató kiválasztotta a megfelelő szakját, valamint tantárgyfelvétel időszaka van és a tárgy előfeltételei teljesítettek, a hallgató felveheti az adott tárgyat, azaz, kiválasztja a kurzust, melyre jelentkezni szeretne.

Tárgy leadása:
A hallgató kiválasztja a megfelelő tárgyat, és ha tárgyfelvételi időszak van, a tárgyat leadhatja.

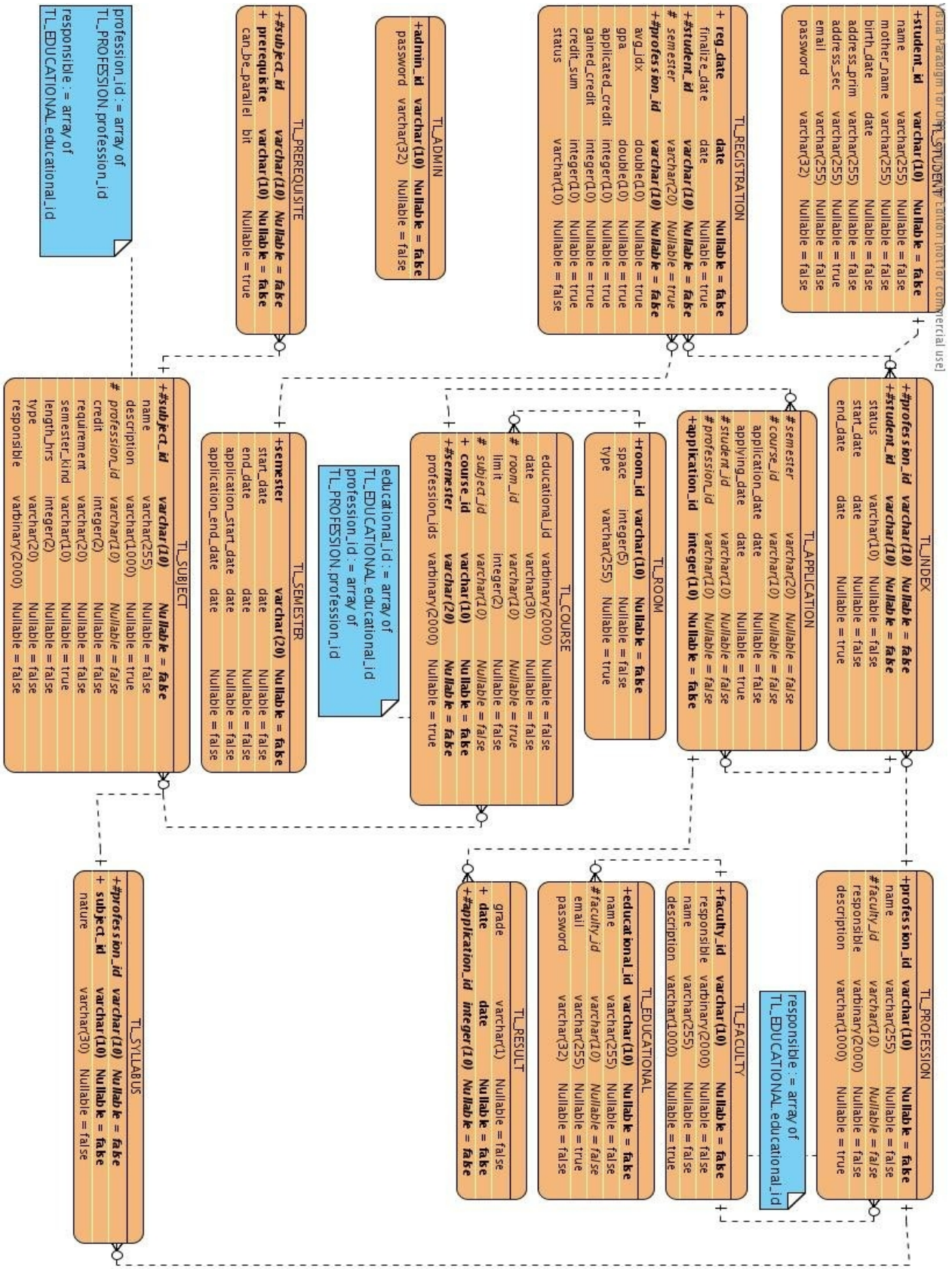
Kurzusváltoztatás:
Amennyiben tárgyfelvételi időszak van, a hallgató egy adott tárgyhoz tartozó másik kurzust kiválaszthat.

Tárgy meghirdetése/meghirdetés visszavonása:
Az oktató kiválaszthat egy tárgyat, és azt meghirdetheti/meghirdetést visszavonhat a saját neve alatt az aktuális félévre vonatkozóan, amennyiben tárgyfelvételi időszak van.

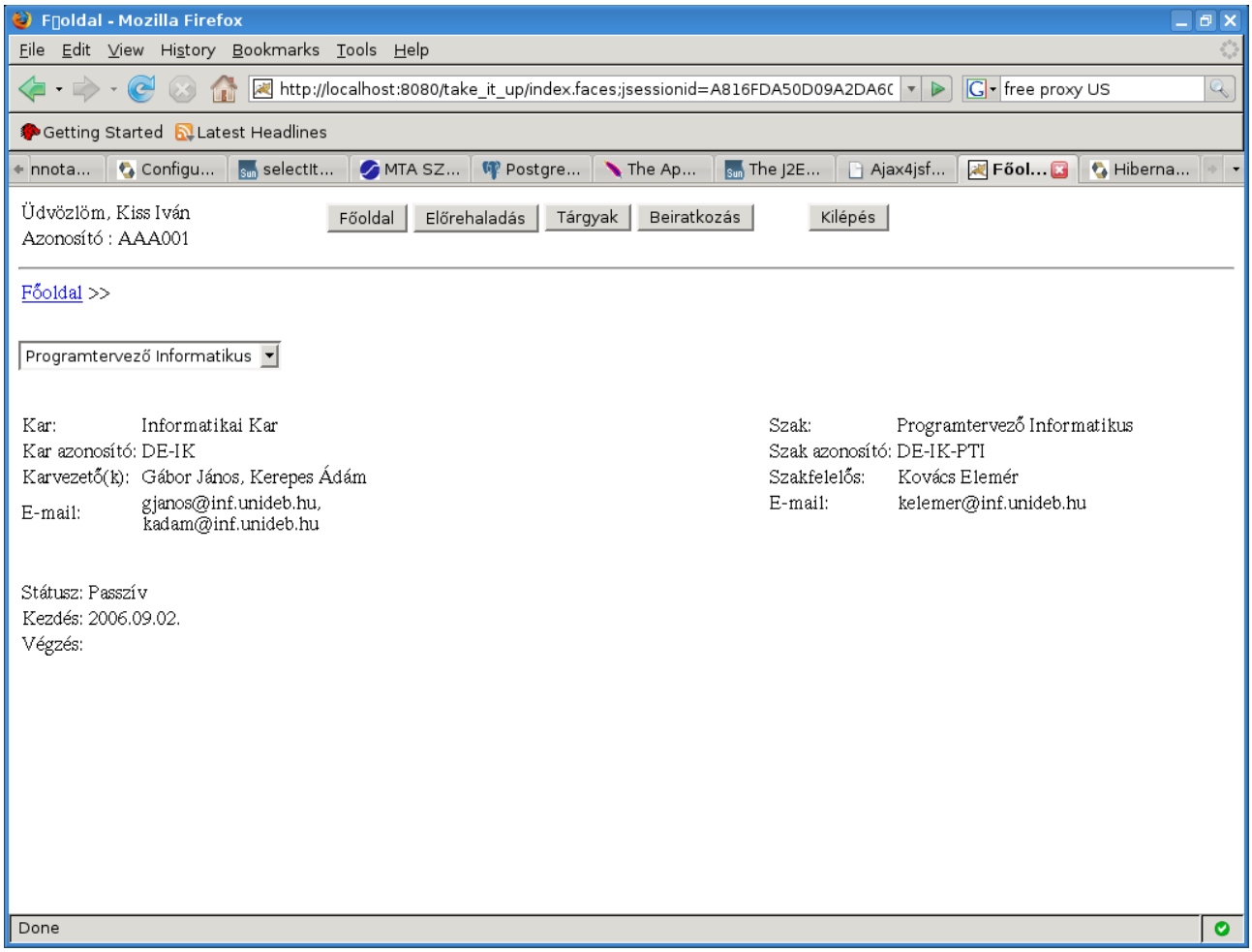
VI.5. UseCase diagram:



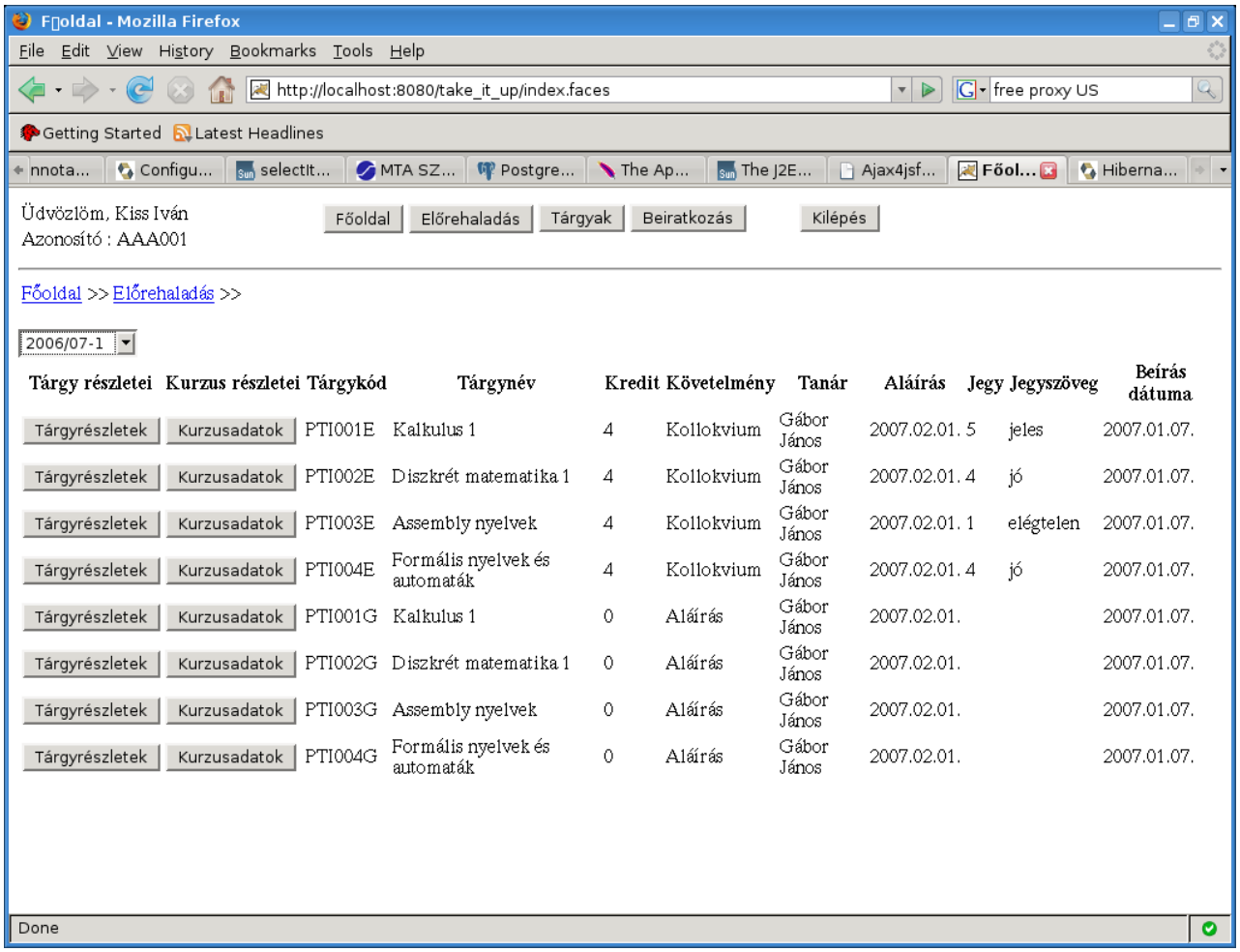
VI.6 Adatbázisterv:



VI.7 Az alkalmazás főoldala:

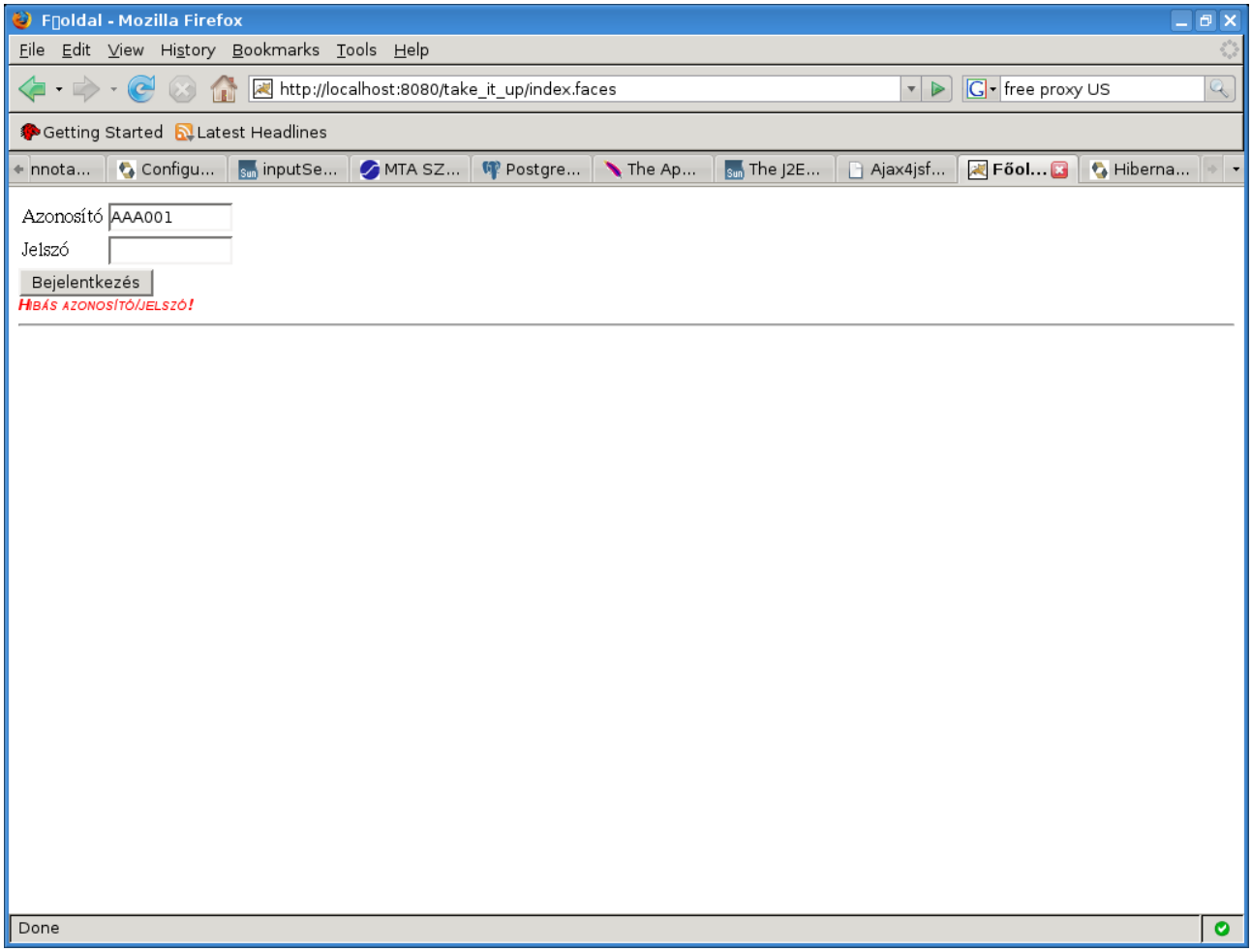


VI.8 Előrehaladás megtekintése:

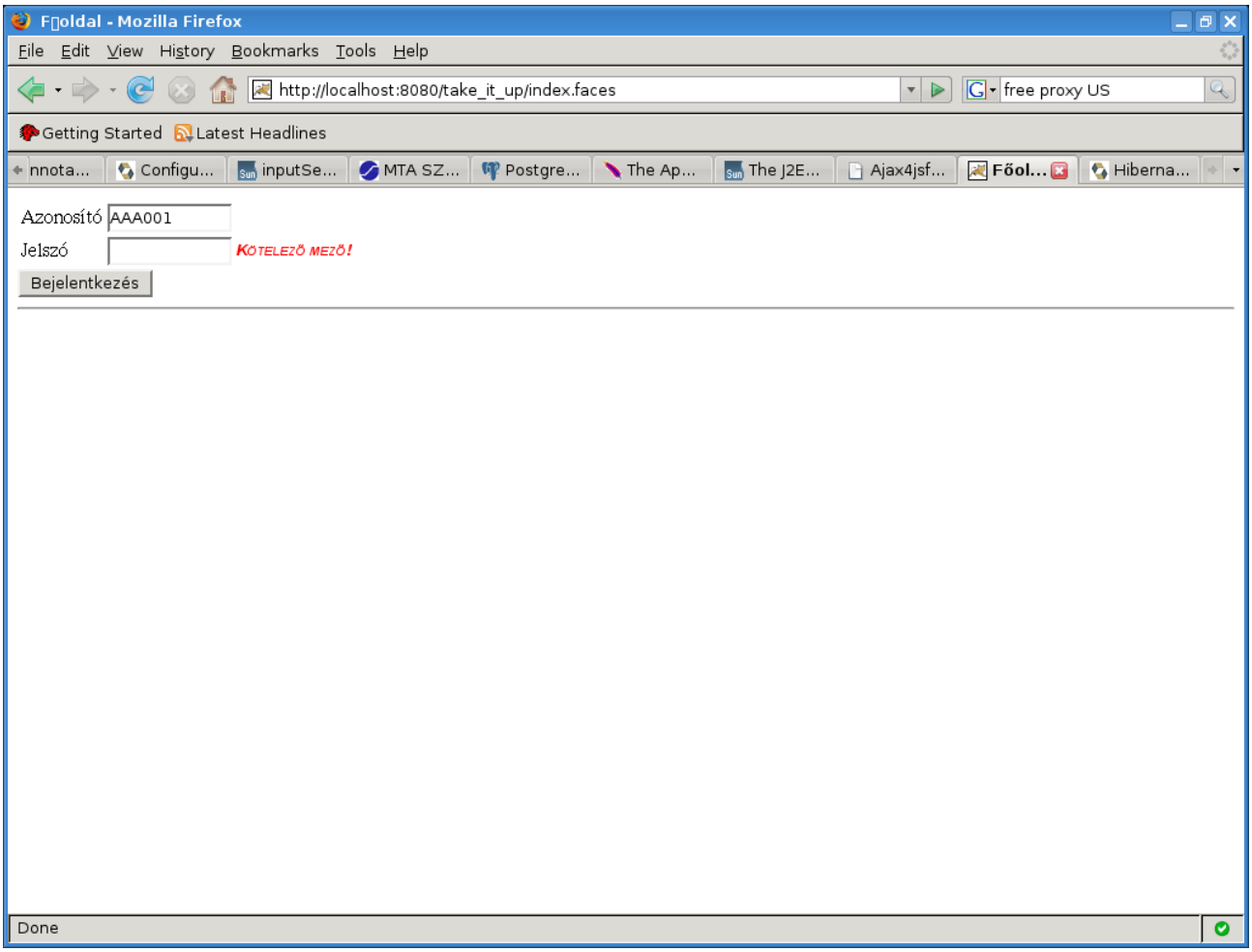


Tárgy részletei	Kurzus részletei	Tárgykód	Tárgynév	Kredit	Követelmény	Tanár	Aláírás	Jegy	Jegyszöveg	Beírás dátuma
Tárgyrészletek	Kurzusadatok	PTI001E	Kalkulus 1	4	Kollokvium	Gábor János	2007.02.01.	5	jeles	2007.01.07.
Tárgyrészletek	Kurzusadatok	PTI002E	Diszkrét matematika 1	4	Kollokvium	Gábor János	2007.02.01.	4	jó	2007.01.07.
Tárgyrészletek	Kurzusadatok	PTI003E	Assembly nyelvek	4	Kollokvium	Gábor János	2007.02.01.	1	elégtelen	2007.01.07.
Tárgyrészletek	Kurzusadatok	PTI004E	Formális nyelvek és automaták	4	Kollokvium	Gábor János	2007.02.01.	4	jó	2007.01.07.
Tárgyrészletek	Kurzusadatok	PTI001G	Kalkulus 1	0	Aláírás	Gábor János	2007.02.01.			2007.01.07.
Tárgyrészletek	Kurzusadatok	PTI002G	Diszkrét matematika 1	0	Aláírás	Gábor János	2007.02.01.			2007.01.07.
Tárgyrészletek	Kurzusadatok	PTI003G	Assembly nyelvek	0	Aláírás	Gábor János	2007.02.01.			2007.01.07.
Tárgyrészletek	Kurzusadatok	PTI004G	Formális nyelvek és automaták	0	Aláírás	Gábor János	2007.02.01.			2007.01.07.

VI.9 Általános hibaüzenet megjelenítése az oldalon:



VI.10 Validációs hibaüzenet megjelenítése az oldalon:



Köszönetnyilvánítás

Ezúton szeretnék köszönetet mondani mindazoknak akiknek munkája, segítsége, tanítása nélkül e dolgozat nem jöhetett volna létre.

Köszönettel tartozom:

Dr. Juhász István Tanár Úrnak a sok ismeretért, tudásért, melyet a 3 év leforgása alatt sikerült megosztania velem, valamint témavezetőként nyújtott segítségéért;

Gábor Andrásnak az átfogó technológiai ismeretekért;

Kollár Lajosnak az adatbázisrendszerek területén nyújtott ismeretekért;

Espák Miklósnak a Java alapjainak elsajátításában nyújtott segítségéért.